GPU HIGH-PERFORMANCE FRAMEWORK FOR PIC-LIKE SIMULATION METHODS USING THE VULKAN® EXPLICIT API

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Kolton Yager

March 2021

COMMITTEE MEMBERSHIP

TITLE: GPU High-Performance Framework for PIC-like Simulation Methods Using the Vulkan® Explicit API

AUTHOR: Kolton Yager

DATE SUBMITTED: March 2021

COMMITTEE CHAIR: Zoë Wood, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Aaron Keen, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Maria Pantoja, Ph.D.
Professor of Computer Science

ABSTRACT

GPU High-Performance Framework for PIC-like Simulation Methods Using the Vulkan® Explicit API

Kolton Yager

Within computational continuum mechanics there exists a large category of simulation methods which operate by tracking Lagrangian particles over an Eulerian background grid. These Lagrangian/Eulerian hybrid methods, descendants of the Particle-In-Cell method (PIC), have proven highly effective at simulating a broad range of materials and mechanics including fluids, solids, granular materials, and plasma. These methods remain an area of active research after several decades, and their applications can be found across scientific, engineering, and entertainment disciplines.

This thesis presents a GPU driven PIC-like simulation framework created using the Vulkan® API. Vulkan is a cross-platform and open-standard explicit API for graphics and GPU compute programming. Compared to its predecessors, Vulkan offers lower overhead, support for host parallelism, and finer grain control over both device resources and scheduling. This thesis harnesses those advantages to create a programmable GPU compute pipeline backed by a Vulkan adaptation of the SPgrid data-structure and multi-buffered particle arrays. The CPU host system works asynchronously with the GPU to maximize utilization of both the host and device. The framework is demonstrated to be capable of supporting Particle-in-Cell like simulation methods, making it viable for GPU acceleration of many Lagrangian particle on Eulerian grid hybrid methods. This novel framework is the first of its kind to be created using Vulkan® and to take advantage of GPU sparse memory features for grid sparsity.

ACKNOWLEDGMENTS

Thanks to:

- Dr. Zoe Wood for supporting and guiding this thesis.
- Dr. Andre Pradhana for his insights into PIC-like methods.
- The KhronosDevs Slack community for their technical support and insights.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

The computer simulation of continuum mechanics concerns a wide variety of materials including fluids, non-rigid bodies, particulate substances, and beyond. All are materials which are characterized by the complex behaviors that emerge as a result of interactions between their infinitesimal constitutive elements. These interactions occur at a scale which is far beyond human perception, and in quantities which exceed the practical limits of computing.

Modelling and simulation of these materials therefore requires a holistic perspective, one which describes the emergent behaviors of the material rather than individual interactions. Materials are conceptualized as a continuum of matter with material properties defined at every point in space where the material exists. Material behavior can then be described with respect to how these material properties vary over time, space, and relative to one another. These descriptions typically emerge as partial differential equations which may be exceedingly complex, and which are difficult or impossible to solve analytically. The study of computational continuum mechanics applies a wide variety of methods in order to discretize and accurately solve these challenging problems. This is typically done with the intent to accurately recreate observable real-world behaviors.

The terms Lagrangian and Eulerian refer to the two common descriptions used when modelling a continuum. The Lagrangian description treats the continuum as a collective of many freely moving material elements. The Eulerian description views the continuum from a fixed external frame of reference, and observes material as it travels through space. In computing, the Lagrangian and Eulerian descriptions are also coupled with two different methods for discretization of a continuum: into par-

ticles and into a grid. Both descriptions are equally applicable, but exhibit different advantages and disadvantages. Likewise, hybrid methods have been developed which attempt to harness the best of both models by transferring information between representations and simulating mechanics on both.

Many computational continuum mechanic methods belong to a category characterized by the tracking of Lagrangian particles over an Eulerian background grid. The conception of these methods began with the Particle-In-Cell (PIC) method, and have since branched into a large field which includes Fluid Implicit Particle (FLIP), Material Point Method (MPM), Generalized Interpolated Material Point (GIMP), and many more. While the specifics of any given simulation varies greatly by the material(s) being simulated, all of these methods are tied together by their Lagrangian/Eulerian hybridization.

The category of methods which have descended from PIC simulation accomplish Lagrangian/Eulerian hybridization by tracking Lagrangian particles, also commonly referred to as 'material points', over an Eulerian background grid. Quantities are transferred between the two representations such that portions of the simulation's computations can be conducted on each. These methods have proven highly effective in the accurate simulation of many substances, and have likewise become the foundation for many applications within engineering, scientific, and entertainment contexts.

In science and engineering, these methods allow for predictive simulations which can reveal the nuanced interactions which might emerge in practice. By allowing these behaviors to come to light prior to fabrication or lab testing, research and development may be greatly accelerated. Computational fluid mechanics, for example, plays a critical role in the design of aerospace vehicles with respect to their aerodynamics.

The entertainment industry, the computer graphics film and visual effects industries in particular, also make extensive use of these hybrid simulation methods. Simulation of continuum materials can result in visually compelling effects by virtue of their complex behaviors. Continuum simulations can also go a long way towards furthering the illusion of life in CG worlds. They allow for presentation of a more dynamic world which reacts to external stimuli in a way which is more consistent with our intuition.

The etymology of 'Lagrangian particle on Eulerian background grid' methods is long and lacking in consensus. Historically, it is commonly accepted that these methods all share a common ancestor with the Particle-in-Cell (PIC) method introduced in the 1960's. However classical PIC simulations can also be accurately described as being a subset of the Material-Point-Method. Meanwhile MPM itself is a subset of the Generalized Interpolated Material Point (GIMP) method model.

To simplify the language of this thesis, we will henceforth use 'Particle-in-Cell like methods' abbreviated to 'PIC-like' or simply 'PIC' methods, to refer to the entire category of simulation methods. Where the category of methods in question is characterized by the tracking of Lagrangian particles over an Eulerian background grid. 'Classical Particle-in-Cell' will be used to make any references to the original PIC methods developed in the 1960's. The reader should expect that related works will not utilize this same language.

Although not always embarrassingly parallel, many PIC-like simulations exhibit a great degree of parallelism at scale and are thus well suited for general purpose GPU (GPGPU) adaptation. The latest generation of graphics and compute API for GPU applications offer a greater degree of control compared to their predecessors. Explicit API, such as the open standard Vulkan®, have enabled new opportunities for exploiting GPGPU compute power by offering lower API overhead and greater

developer control over resources and scheduling. In this thesis, we will utilize these offerings to produce a unique high-performance GPU compute framework for PIC-like simulations.

1.0.0.1 Contributions

This thesis presents a Lagrangian Particle on Eulerian grid hybrid framework implemented using the Vulkan® API conducted by a C++ host program. The framework provides a consistent and type agnostic interface for accessing both grid and particle resources from GPU compute code. These interfaces communicate with and are configured by the host program in such a way that resource management is largely abstracted away from GPU code.

This thesis' most significant and highlighted contributions are a Vulkan sparse paged grid implementation (Chapter 5), a programmable GPU compute pipeline for PIC-like simulations (Section 4.1), high-throughput multi-buffered particle arrays (Chapter 6), and an asynchronous simulation loop (Chapter 7).

The summary result of these contributions is a high performance GPU compute framework for PIC-like simulation. It exhibits high GPU and CPU utilization, high throughput, and the programmable flexibility to support a wide range of PIC-like simulation methods.

We test our sparse grid implementation with unit and coverage tests validating its function as a general purpose GPU sparse grid. We validate the whole framework with a non-physically based PIC-like simulation pipeline which utilizes all of the key processes of PIC-like simulation. We observe an up to eight times increase in overall simulation speed as a result of our asynchronous simulation loop and particle multi-buffering. We also demonstrate our GPU sparse grids ability to support grid dimensions which exceed the total physical memory on our available GPU hardware.

Chapter 2

BACKGROUND

2.1 Lagrangian Particles and Eulerian Grids

Defined more generally than in the previous paragraphs, the Eulerian and Lagrangian perspectives are not a simple matter of grids and particles. The Eulerian description, as defined in continuum mechanics, is a description which models the continuum from a constant frame of reference. The frame of reference does not follow the deformation of continuum bodies, instead allowing the material to to travel through the Eulerian frame[26, 7]. Regular Cartesian grids are often used to discretize space in Eulerian descriptions as they are intuitive and simple to implement. In such an application, the Cartesian grid subdivides space into a discrete number of rectilinear 'cells' or 'nodes'. Each cell contains, or more accurately observes, continuum material as it travels through space.

The Lagrangian description utilizes a changing frame of reference. The frame of reference follows bodies as they undergo deformation[26, 7]. In the case of Lagrangian particles, each particle is a discrete representative of the larger continuum material which travels through space as the material deforms. Each particle can be associated with any number of continuum variables which represent the state of that portion of the larger continuum material body. By virtue of their individual movements through space, each particle represents a unique frame of reference from which to observe the continuum.

The costs and benefits of both descriptions can vary depending on the mechanics being modelled. In some cases it may come down to a difference in the simplicity of

formulae when mechanics are modelled in one description versus the other. However there are some commonly acknowledged strengths and weaknesses.

The most commonly cited virtue of the Lagrangian description is its ability to support history dependent mechanics[50, 51, 17, 7, 9]. The mechanics of many materials are relative to the material's prior state or rest state. Elasticity is a common example of this type of mechanic. Since the Lagrangian description individually tracks finite material units through all deformations, there is no ambiguity in their history. In the Eulerian description, these mechanics are more difficult to model. Each grid cell is defined by the state of all material contained within regardless of that material's origin. When undergoing large deformations, it is likely that quantities of material with vastly different initial state will eventually occupy the same grid cell. When this occurs the distinctiveness of those material quantities can be obfuscated, or lost entirely.

Lagrangian particles also excel at maintaining a clear separation between materials in a simulation. Since each particle is uniquely identifiable and possesses distinct material information, mixture between materials can occur without changing their representation. This benefit also applies to the tracking of material interfaces[17, 9, 10, 15]. In an Eulerian description, two or more unique materials being present within the same Eulerian node may be poorly defined or result in irreversible mixture[26].

However a grid representation can better handle large deformations in some cases, as a result of their unchanging frame of reference. They are also highly effective for the computation of gradients, neighborhood evaluation, and other finite differencing schemes[50]. Evaluation of adjacency on a grid is simple conceptually and computationally. However Lagrangian particles can be distributed through space arbitrarily. Some particles may be separated from their nearest neighbors by very large distances, while others are tightly packed into clusters. Evaluation of material point variables

relative to the state of their neighbors is both inconsistent and computationally challenging.

Overall, the Lagrangian representation is unchallenged in its ability to holistically represent the state of continuum materials in-space. Likewise most PIC-like methods have become 'full-particle' methods, in which the particle representation of materials is the dominant format. In such methods, the grid exists as a scratchpad for grid calculation, and only particle data carries the full state of the simulation materials. Despite the diminished role of the grid, these methods still represent a Lagrangian Eulerian hybridization, and require both particle and grid resources in implementation.

2.2 Particle-In-Cell Simulation

The core of PIC-like methods is the transfer of information between a Lagrangian particle representation and Eulerian grid representation. In modern parlance, this transfer is commonly referred to as the 'Particle To Grid' and 'Grid To Particle' stages and are abbreviated as 'P2G' and 'G2P' respectively. The details of how information is transferred between representations can vary, and is often part of what sets one PIC-like method apart from another. However P2G always involves an interpolation of particle variables into surrounding grid cells. Some methods use a similar transfer of grid variables to particles within their G2P stage, while others instead advect particle positions using grid variables then discard the grid values. This is illustrated by Figure 2.1 and Figure 2.2.

The computations done on both particle and grid data in-between transfer stages depend entirely on the material being simulated and the constitutive model being applied. The formulation of constitutive models and their translation into code is an expansive subject outside the scope of this thesis. For the purposes of this write-up,

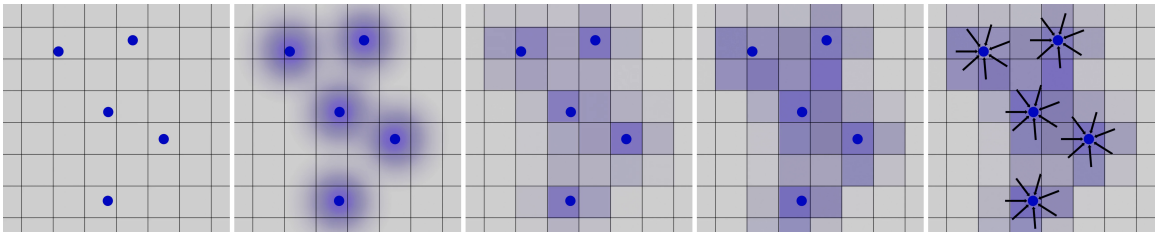


Figure 2.1: Illustration of PIC-like simulation transfer processes in an implementation which reconstitutes particle state from values computed on the grid.

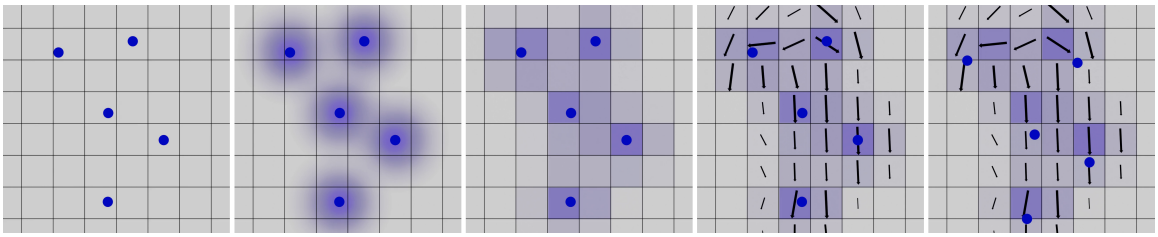


Figure 2.2: Illustration of PIC-like simulation transfer processes in an implementation which advects particles using a vector field computed on the grid.

it should be sufficient to know that most constitutive models are expressed in the form of partial differential equations, which are then typically solved using numerical methods. Implementations must also choose between using explicit or implicit methods to find solutions for their constitutive model. The benefits and costs of each are often weighed in relation to the constitutive models in use, and the intended use cases of the PIC-like simulator being implemented.

Classical PIC is the earliest PIC-like method. It was developed at the Los Alamos Laboratories in the late 1950's and 1960's for the study of compressible fluid mechanics. It introduced the combination of particle and grid data as a compromise which would allow large distortions in fluid to be handled well on the grid, while the particles provided a full history for each fluid element[26]. Classical PIC has since been replaced by newer methods in many fields, but found particular success in the study of plasma, where it remained relevant into the 2000's[10, 52, 19].

The Fluid Implicit Particle method (FLIP) was introduced in the 1980's as a PIC-like method which was intended to more effectively handle fluid flows. One of FLIP's key goals and contributions was the reduction of the dissipative effects present in PIC methods of the time. The identified the transfer of attributes between the grid and particles as the primary source of the undesired diffusion. Their solution was to make the simulation more Lagrangian by updating particle attributes using grid information as opposed to fully re-writing particle attributes from interpolated grid data[10].

Inspired by prior successes and the history preservation of particle methods, PIC-like methods were eventually applied to solid-mechanics[51, 15, 9]. Early papers on this work are considered to be the origins of what is now known as the Material-Point-Method. This work and the work that followed served to expand PIC-like methods to cover simulation of all types of continuum materials and mechanics. These methods have since continued to develop up into the present with regular introductions of new constitutive models, solvers, data-structures, and transfer methods.

2.2.1 Sparse Grids

The most intuitive way to implement an Eulerian description of continuum is with a regular Cartesian grid. The resolution of a grid can be conceived as the result of recursively subdividing the simulation domain. Each cell in the resulting grid possesses an identical set of attributes describing variables of the material(s) and ambient space. In code, this can be simply written as a fixed sized array of 'cell' structures, or multiple fixed sized arrays of cell attribute values. However such an implementation suffers greatly from growth in complexity, especially in 3D where complexity growth is cubic. Cartesian grids can quickly consume all available physical memory after several subdivisions, and any algorithm which must operate across the

full grid will also exhibit cubic complexity growth. Furthermore, the entire grid domain may not always be utilized, in which case increasing grid resolution will provide diminished benefits relative to cost.

Sparse grid data-structures tackle these practical implementation challenges while conceptually maintaining the intuitive Cartesian grid. In recognition of the fact that often only a small portion of the simulation domain is occupied with "interesting" information, sparse grids endeavour to allocate for and process only the occupied regions of a grid. The exact mechanisms used to accomplish this type of sparse grid data-structure vary, as will be discussed in Chapter 3: Related Work. Our implementation is a GPU version of the *Sparse Paged Grid*[46] data-structure which is characterized by its use of virtual memory as the basis for grid sparsity.

2.3 Introduction to the Vulkan® Explicit Graphics/Compute API

Vulkan® is a combined graphics and compute API which grants developers explicit control over many GPU resources. Vulkan is an open standard maintained by the Khronos group and it is intended primarily for the creation of cross-platform high performance computer graphics. The API also supports general purpose GPU computation making it a viable tool for cross-platform high performance computing. Aside from being platform independent, Vulkan appreciates lower overhead relative to its predecessors, support for asynchronous API usage, predictability, and the tunability enabled by explicitness [36].

Vulkan® was chosen for this thesis primarily for its technical advantages, but also due to the author's prior experience with the API. At present, this project uses only the compute capabilities of Vulkan. In the future the ability of Vulkan to share resources between graphics and compute contexts may be advantageous for providing a real-time view of simulations. We are also interested in the possibility of expanding

our implementation to utilize multiple GPU heterogeneous or otherwise. In both cases Vulkan provides a unified interface to all devices and could be well suited for scaling across a distributed system.

2.3.1 The Basics

Vulkan® is a large API, whose specification documentation is notoriously difficult to absorb. Likewise we will introduce only a subset of the API's concepts which are necessary to understanding this thesis. At the highest level, use of the Vulkan API begins with the creation of a Vulkan instance which links a Vulkan application to the Vulkan implementation. Although many applications will only ever create and destroy a single Vulkan instance, it is not a global state, and multiple instances may exist within the same application. Instances are also configured with the API version, optional instance level extensions, and optional layers used mostly for debugging and profiling.

Within an active Vulkan instance, a system's physical GPU devices can be queried. Vulkan provides detailed information about the capacity and capabilities of each physical device. The application is able to validate any assumptions about features it will require by checking the properties of the reported physical devices. Assuming compatible device(s) are found, the application then creates one or more 'logical device' objects. Vulkan logical devices provide a pre-configured interface to a physical device. They are created with explicit specification of any device level extensions or optional features required of the device, and a successfully created logical device guarantees support for these features and extensions. The majority of remaining API calls operate with/on individual logical devices to affect the corresponding physical device.

One of the most critical properties of Vulkan physical devices is their enumeration of 'queue families'. Each queue family available from a device may have distinct capabilities. Capabilities are indicated by a union of the following flags: `VK_QUEUE_GRAPHICS_BIT`, `VK_QUEUE_COMPUTE_BIT`, `VK_QUEUE_TRANSFER_BIT`, `VK_QUEUE_SPARSE_BINDING_BIT`, and `VK_QUEUE_PROTECTED_BIT`. The first three represent a core set of capabilities, allowing use of the graphics pipeline, general purpose computation, and GPU memory operations respectively. `VK_QUEUE_SPARSE_BINDING_BIT` indicates a queue families ability to manage the binding and unbinding of memory pages to virtual address spaces. As is detailed in the implementation section, the presence of multiple queue families with overlapping capabilities may indicate a device's support for asynchronous workloads.

A Vulkan 'queue' is the object to which batches of GPU work are submitted. Work submitted to queues is started in the same order with which it is submitted, but may execute concurrently and complete out of order unless additional synchronization is specified. Additionally, work can be submitted to multiple queues as a way to submit tasks which may be submitted or executed asynchronously[1].

Almost all work submitted to queue's come in the form of 'command buffers'. Command buffers are an object containing a pre-recorded sequence of commands which might alter the GPU state or initiate work such as a memory transfer, draw call, or compute dispatch. The execution of command execution can be controlled using pipeline barriers. These barriers are scoped to specific GPU operation types such as the execution of fragment shaders or memory transfer, and they establish explicit dependency relationships between the memory resources utilized by commands. The command buffer model allows for GPU workloads to be setup in advance of their

[1]Although permitted by the specification, it is atypical for queues from the same queue family to execute asynchronously. It is more typical that they simply process submitted workloads asynchronously.

execution, minimizing overhead by allowing the GPU to execute many operations without intervention by the host.

2.3.2 Resources

For the purposes of this thesis, nearly all GPU resources will be Vulkan buffer objects (`VkBuffer`) and the memory spans (`VkMemory`) backing them. When accessed through commands or shader code, Vulkan buffers behave as contiguous spans of memory. However, each buffer object is created without any physical memory, and must be manually bound to a span of previously allocated memory. Vulkan devices typically expose many different memory heaps with specific properties and capacities. It is the responsibility of developers to select the appropriate heap for each resource. Vulkan also expects developers to do their own memory management, so memory is typically requested from the API in large chunks, then sub-allocated to an application's resources. We partly outsource this responsibility to the *Vulkan® Memory Allocator*[6] library.

Vulkan buffers may exhibit different capabilities and performance depending on the type of memory by which they are backed. To simplify these differences, we commonly describe buffers as being one of two types: 'Device Local' or 'Host Coherent'. Device Local refers to buffers backed by memory which is resident to the GPU. It should be assumed that these buffers are optimal for direct access from the device, but cannot be read/written by the host. Host Visible refers to buffers which can be directly accessed by the host system through memory mapping. These buffers are necessary for transferring data between the host and GPU, but access to these buffers on the GPU may be less efficient than device local buffers.

The final Vulkan resource type which needs introduction is sparse resources[2]. Vulkan supports both sparse buffers and sparse images, but only buffers are relevant to this thesis. Sparse buffers behave like other buffers when accessed from GPU code, but unlike their non-sparse counterpart, sparse buffers do not need to be backed by a contiguous span of memory. Instead disjoint pages of memory are bound to sub-ranges of the sparse buffer. With the further allowance of the sparse residency feature, sparse resources do not need to have their full range backed with memory prior to usage. As a result, sparsely resident buffers are virtual address spaces both in effect and implementation. This fact is foundational to our VSPgrid implementation, as will be explained in Chapter 4.

2.3.3 Synchronization

The Vulkan® API's emphasis on asynchronous design and host concurrency necessitates the existence of many synchronization primitives and functions. These synchronization tools, like the specification itself, are notoriously arcane. We will be leaving many details of Vulkan synchronization out of this write-up, but they must be discussed if our implementations asynchronous nature is to be understood.

Most choices of synchronization primitives are made based on where, when, and how the primitive will be signalled, waited-upon, or reset. This thesis uses binary semaphores (`VkSemaphore`) to create dependency relations between batches of work submitted to one or more Vulkan queues. We uses fences (`VkFence`) to pause host execution until the GPU completes one or more tasks. Events (`VkEvent`) are also used to synchronize host actions, but with finer grain as events can be signalled from within an executing command buffer. Finally, pipeline barriers (`VkCmdPipelineBarrier`) are used within command buffers to prevent read/write hazards between GPU operations

[2]Sparse resources and sparse residency are optional features not supported on all hardware. However, most modern discrete GPU support both.

Primitive	Sync Scope	Description
`VkSemaphore`	Device	Signalled and waited-upon by batches of work submitted to queues.
`VkCmdPipelineBarrier`	Device	Inserted into command buffers to prevent concurrent execution of dependent commands.
`VkFence`	Host ↔ Device	Coarse synchronization between host and GPU. Fences are signalled by a queue when work completes. Waited upon and reset by host.
`VkEvent`	Host ↔ Device	Fine grain synchronization between on GPU. Can be modified and polled by the host, or by command buffer commands.

Table 2.1: Vulkan synchronization quick reference. Each primitive is described only as it is applied within this thesis, and does not reflect all use-cases.

which modify GPU memory. A quick reference for these use-cases is given by Table 2.1.

2.3.4 SPIR-V and GLSL

OpenGL, Khronos® group's previous generation graphics API, defined its own high level shader language. The OpenGL shader Language (GLSL) provided a platform agnostic C-like shader language requiring OpenGL implementations to handle GLSL compilation at run-time. When developing Vulkan, and other API in the latest generation, it was decided that a high-level shading language should be foregone in favor of a lower-level intermediate representation which could provide more predictable cross-platform behavior and simplify run-time compilation. SPIR-V is the intermediate representation created for/chosen by the Vulkan API.

SPIR-V is an intermediate representation originally built on the LLVM intermediate representation. It has since evolved into its own specification, but remains closely adjacent to LLVM, and uses the same SSA form. SPIR-V is designed for represen-

tation of parallel compute instructions in general, and is not restricted to a specific API or hardware architecture. Developers using the Vulkan API provide both their graphics shader code and GPU compute code through binary SPIR-V modules. Each module is eventually passed through to the device driver for final configuration and compilation into device native instructions.

SPIR-V is the only GPU code representation that the Vulkan API ingests. Although it is possible for developers to manually write their SPIR-V modules, it is common for Vulkan developers to write shaders using a high-level shading language which is then compiled into SPIR-V before run-time. With this expectation, GLSL has been expanded for use with the Vulkan API via compilation into SPIR-V. This shift has allowed shader language tooling to develop significantly, and introduced tools like *Shaderc*[4] which provide a suite of tools for compiling high-level shader languages into SPIR-V, as well as code optimization, cross-compilation, and reflection. This thesis writes its GPU compute code using GLSL, and compiles that code using the Shaderc project's 'glslc' tool. Use of this tool is necessary to enabling the creation of a PIC resource interface as described in Section 4.1.2.

Chapter 3

RELATED WORKS

Throughout their development, PIC-like simulation methods have consistently proven their effectiveness at realistically representing an impressive range of real world materials, their mechanics, and interactions. This versatility has inspired application of PIC-like methods in science, engineering, and entertainment. Granted multidisciplinary relevance, continued research into PIC-like methods is granted importance as well as a more diverse set of challenges to undertake. With PIC-like simulation being implemented in more varied contexts, the variety of opportunities for scientific and engineering contributions expands as well.

We would label research contributions relating to PIC-like simulation as members of three categories. The first is the invention and improvement of solvers used in PIC-like simulations. Solvers refers to the component of a simulator which analyzes present state and solves for future state using the physically based formula of the constitutive model. Solvers, PIC-like or otherwise, may make use of any number of numerical or analytical methods to achieve their goal. As a result, solver design represents a large portion of research on PIC-like simulation. Decisions made in designing a solver may profoundly affect simulation speed and accuracy, and their efficacy and performance may also vary in the face of the different material models being implemented. It may therefore be justified that solver designs be significantly specialized to fit the needs of a particular industry or use case.

The second category is the development of new material models. Recall that material models, also called constitutive models, are the physically based mathematical models describing the mechanics of a simulated material. While PIC-like methods have proven themselves extremely potent in the simulation of varied materials, many

of these materials will nonetheless require a specialized material model to re-create their unique mechanics. Research in this category typically contributes novel material models for realistic simulation of materials or material interactions not previously achieved. Some examples from recent years include the dynamics of bread[54], solids becoming molten[22], and foams[55].

The third and final category of contributions covers computational methods and software engineering. Alongside all of the purely mathematical and physical aspects of PIC-like simulation, is a need for well developed software. The design and usage of data-structures and algorithms for PIC-like simulation can have a massive impact on the ultimate effectiveness of the method. Research within this category frequently proposes alternative data-structures and algorithms which drive PIC-like simulations, or introduce novel abstractions of existing methods which may allow for easier adaptation of existing technologies to new materials and solvers.

More recently there have been many efforts to develop PIC-like simulators which utilize massively parallel computing. More than just re-writing existing solvers on new platforms, application of PIC to GPU, multi-processor, and distributed systems requires fundamental rethinking of the way data flows through a simulator application. This thesis falls squarely into this category through its contribution of a Vulkan based GPU PIC-like simulation framework.

Being a GPU focused thesis, our review of related works will be biased towards similar contributions from other researchers. We will provide a brief historical overview of the development of PIC-like methods. Afterwards we will switch to a discussion of sparse grid data-structures, followed by a review of existing GPU based PIC simulation research. We end our related works with a discussion of existing applications of the Vulkan API for non-realtime GPU computation problems.

3.1 The History of Particle-In-Cell Methods

Particle-In-Cell simulation was originally developed at Los Alamos labs in the early 1960's. The formative paper on the subject is *The Particle-In-Cell Method For Numerical Solution of Problems in Fluid Dynamics* written by Francis H. Harlow[26]. The work introduces the foundational elements elements of PIC-like Lagrangian Eulerian hybrid simulation, and proposes a model for fluid simulation. Despite the ongoing prominence of PIC-like methods, classical PIC has largely been surpassed by its own descendants. However it has maintained a more lasting relevance within the sub-field of plasma simulation [52, 19, 41].

The next landmark in the development of PIC-like methods is the advent of FLIP in 1987[10]. The Fluid Implicit Particle Method (FLIP), is an alternative to classical PIC which continues dual use of particles and grids, but puts a greater emphasis on the Lagrangian particle representation. The primary motivation for FLIP was to avoid the dissapative effects of classical PIC. In classical PIC, both the particle and grid data-structures will, during computation of a simulation frame, contain a comprehensive representation of some state of the simulated materials. Transfers between representations involves a reconstitution of new values. This repeated reconstitution of values between representations is a primary source of dissipation in PIC simulations.

FLIP reduces this effect by allowing only particles to store the complete state of simulated material. Instead of conducting a full transfer of values, a subset of values is transferred to the grid, solved on the grid, then used to update particle values rather than replace them. In particular, FLIP solves an acceleration field on the grid, then uses the resulting accelerations to update Lagrangian particle velocity and position[10]. Many modern PIC-like methods are similar to FLIP in their use of particles as the canonical representation of material state.

With the success of PIC and PIC-like simulation techniques in fluid simulations, the method was eventually extended to solid mechanics as well. By this time, PIC-like simulations become a more general purpose method for simulation of continuum materials. The work of Deborah Sulsky et al [50, 51] in the mid 1990s has been cited as the origin of the Material Point Method (MPM)[22]. Although referencing PIC in the title, Sulsky's application of PIC methods to solid mechanics is more specifically derived from FLIP, storing no fully representative information in the grid. Sulsky also makes particular note of PIC-like simulation's advantage in implementing history-dependent mechanics[50].

Papers following the initial application of PIC to solid mechanics further grew the techniques applicability to non-fluid materials. This research includes the study of granular flow, a material whose simulation continues to be the subject of research today. Researchers also innovated by experimenting with new types of PIC-like solvers in an attempt to find more efficient and numerically stable solutions. Cummins et al is an example of both. Although not the first to apply MPM to granular material flow, they were the first to develop an implicit formulation of MPM for the problem[15].

Bardenhagen et al[9] further clarified PIC-like methods as a category of simulation methods distinct from other mesh-free methods. Their paper is largely a formalization of many aspects of PIC-like simulation introduced previously, and attempts to create a generalized model for PIC-like methods under the name Generalized Interpolation Material Point (GIMP) methods. The authors further investigate the stability of existing MPM methods on finite deformations, and conclude that the C^1 interpolation kernels typically used for MPM are insufficient. Under the expanded definition of GIMP, they introduce a PIC-like simulation which uses the next level of smoothness in their interpolation kernels to achieve greater numerical stability and physical accuracy.

While there has been no shortage of PIC research since the introduction of GIMP, much of that work focuses more heavily on material models, computing techniques, and solver design. However, the relatively recent introduction of the Affine Particle-In-Cell Method (APIC) deserves to be discussed alongside other direct contributions to the core methods of PIC-like simulations[33].

APIC's significance to the study of PIC-like methods is highly comparable to that of FLIP. Just like FLIP, APIC's primary contribution is an improvement to the particle and grid transfer stages. APIC improves upon FLIP methods by modelling particle velocity as locally affine rather than locally constant[33]. Doing so, and reformulating transfer functions accordingly, results in a method which is more similar to classical PIC, but which preserves angular momentum and avoids dissipation to a degree similar to FLIP. APIC has been adopted by several recent PIC-like implementations, and has been expanded upon by other researchers[22, 34, 31, 37].

PIC-like simulation is a long lived and well studied subject. There have been many publications of all types which will go unmentioned by this thesis. The chronology of PIC is necessary to understand the method and its importance, but this thesis' contributions view PIC-like simulation from a high-performance computing and data-structures perspective instead. Although historically, the majority of PIC-like simulation research has emphasized physics and mathematical methods, there has recently been a surge of interest in the improvement of PIC through better computing[32]. Central to the design of any scalable high-performance PIC-like simulation is the sparse grid data-structure.

3.2 Sparse Grid Data-Structures

A dense Cartesian grid is the natural spatial data-structure for Eulerian representation of continuum. Furthermore, grids, of any dimensions, can be implemented

trivially with a single array in most programming languages. Yet at scale, dense grids quickly become prohibitively expensive, especially in 3D.

Pushing the limits of simulation fidelity frequently requires high resolution grid computations, and naive grid resolution scaling creates a restrictive upper bound on quality. However, it is not uncommon for much of the space allocated for a dense grid to go unused. Simulated materials rarely fill the entire bounding box of a dense Cartesian grid. In PIC-like simulations, the grid is an ephemeral scratch pad for Eulerian continuum computations. Grid data is not only short-lived, but needed only where material points are present. Likewise, paying the full cost of a dense grid spanning the simulation domain is wasteful and will ultimately limit the quality bounds of grid computations.

Sparse grids are a class of data-structures which substitute for dense grids while only paying a fraction of the cost. This is typically accomplished by allocating grid memory selectively, to support the subset of grid cells which contain necessary and/or unique information. At the cost of losing the unrivaled simplicity of the dense grid, a sparse grid achieves far better best and average case complexity in memory and cell-wise computations.

The study of sparse grids has historically been coupled with the study of level-sets, level-sets being a subcategory of grids with a multitude of applications within simulation. Early work on sparse grids and level-sets was dominated by hierarchical data-structures which refined grid level of detail and allocation requirements adaptively. Commonly, quadtrees and octrees were used as to define the spatial hierarchy with node types and subdivision rules varying depending on the application[49, 48, 20, 38].

A simple octree implementation of a level-set would use 3-color octrees in which tree nodes indicated that their covered grid region was either inside, outside, or interface. The first two requiring only a single value to indicate the state for the

covered grid region. The interface nodes allocated space for fine detail grid data, and contain signed-distance values for the space immediately surrounding the level-set boundary.

The work of Frisken et al[20] showed that these methods can be greatly improved with the addition of more expressive node types. They introduced 'Adaptively Sampled Distance Fields', a spatial data-structure agnostic method for representing shape through distance fields. Their application of ADFs to octrees results in a sparse level-set representation which more effectively compresses level-set detail. In particular, their use of a bilinear approximation for the boundary region of level-sets allows for smooth shapes to have their interfaces represented at far lower cost.

Other sparse grid implementations have used compression as a method for reducing redundancy and over-allocation of grid memory. Run-length encoding (RLE) has been used as a straightforward strategy for eliminating the need to explicitly store values for homogeneous region of grid memory. In the case of Curless et al.[16], RLE was employed directly to runs of data in an otherwise dense grid representation. Later research by Houston et al would both augment volumetric RLE as well as apply it alongside other methods for structuring and compressing grid data[30, 29].

In their doctoral dissertation[11], Robert Bridson critiques both standard octree level sets and RLE level sets. Bridson notes that lookup into octrees can be expensive as a result of many pointer operations and cache incoherence. He also finds octrees to be incompatible with certain numerical methods unless additional steps are taken to ensure a wider region of fine boundary are stored. Bridson criticizes RLE methods for being generally incompatible with non-structured operations such as random lookup. Houston's work on alleviating these weaknesses had not yet been published.

Bridson's alternative sparse level-set utilizes a tree-based spatial hierarchy, but one which differs significantly from typical octree level-sets. Bridson's tree has a higher

branching factor than an octree, and limits the depth of the tree such that all dense grid leaf nodes exist at the same level. This construction creates a tree which is much shallower than an equivalent octree, amortizing the cost of random access operations. Leaf nodes are allocated at a fixed size of 5x5x5, a dimension intended to optimize cache performance when operating on the dense grid data at leaf nodes. Elements of Bridson's design are echoed later in level-set history by the VDB data-structure.

Bridson was not the only researcher to critique the use of octrees for sparse level-set representation. Nielsen and Museth cite similar complaints about slow access in the problem statement preceding their presentation of Dynamic Tubular Grid[44] (DT-grid). DT-grid is a significant departure from other sparse data-structures discussed thus far. It shares ancestry with some older tubular methods for level-sets, in which data is stored only within a fixed radius tube around the level-set boundary[45].

DT-grid is strictly non-hierarchical, instead achieving sparsity by compactly storing only the dense data within the level-set tube. Rather than using a spatial hierarchy to accelerate access, DT-grid uses index based connectivity information between its components to achieve $O(1)$ sequential and neighbor access, and logarithmic random access. DT-grid also benefits from being completely boundless in definition, meaning that level-set data could be represented at any point in space without requiring any structural changes to the DT-grid.

DT-grid was later iterated on in Houston et al[29], in collaboration with the DT-grid authors. Hierarchical RLE Level-Set (H-RLE) combines Houston's prior RLE work with the broad benefits of DT-grid. H-RLE uses the dimensionally recursive definition of DT-grid, but applies the RLE scheme from [30] to compress redundancies. H-RLE ultimately presents a robust representation and tool-set for level-sets able to outperform its contemporaries in memory footprint and algorithmic performance.

VDB, commonly known for its open source implementation *OpenVDB*[43], is a giant in the world of sparse grid data-structures. Presented academically by Ken Museth in 2013[42], VDB is a robust data-structure capable of sparsely representing arbitrary 3D volumetric data including level-sets. Boasting complete dynamism of volume values and sparse topology, $O(1)$ complexity for random, sequential, and stencil accesses, and much more, the VDB data-structure and accompanying tool-set were an unprecedented advancement of sparse grid and level-set technology. OpenVDB has since gained widespread adoption within academia and industry, and experienced continual growth as an open-source library and tool-set.

The VDB data-structure is tree based, but differs greatly from prior octree based methods. Similar to Bridson's proposed level-set data-structure[11], VDB combats the cost of traditional spatial trees by using a high branching factor and limiting tree depth. As in Bridson's case, this creates sparsity, but limits tree traversal cost to a small constant. Tree traversal and algorithms on VDBs are also accelerated by VDBs unique and highly compact method for representing tree topology.

VDB trees consist of three distinct node types, each existing at specific tree levels. The 'Root Node' of any VDB exists as a single node at the top of the tree. Unlike the other nodes, the VDB root node stores references to its children in a generic map data-structure, typically a red-black tree or hash-table. The map is keyed by locations in the, conceptually infinite, index space of the VDB and directs to interior nodes. Use of this mapping allows for VDBs to be practically infinite in scope, without sacrificing sparsity. The children of the root node are capable of covering massive regions of the VDB index space, and since most volumetric data will likely be found in localized pockets, the spatial and access complexity of the root-node's map is typically exceptionally low.

All non-root and non-bottom levels of the VDB tree are represented by 'Interior Nodes' which cover increasingly fine sub-regions of space. Interior nodes not only direct down the tree towards denser grid data, but can themselves represent grid data by acting as 'tile' values which indicate when a large chunk of grid space contains a single homogeneous value. Interior nodes efficiently store their child topology through a pair of bitmaps indicating which grid sub-regions are represented by child nodes or tile values. These bitmaps allow for rapid evaluation of topology during traversal, and accelerate sequential and stencil operations by providing quick iteration over active regions.

At the bottom level of the tree are 'Leaf Nodes' containing dense grid data in constant size blocks. Like their parents, leaf nodes use bitmaps to indicate whether or not individual grid cells are active or inactive, but will typically densely allocate the data regardless. VDB also allows for alternative leaf node modes in which leaf data might be compressed, or sourced from an out-of-core data-stream.

The VDB data-structure and tool-set also utilizes an extensive caching scheme to make access to grid elements amortized $O(1)$ on average. In sub-optimal scenarios where full tree traversal is required, mapping of index space coordinates to nodes or leaf data indices is accomplished through low-impact hash/index computations consisting of just a handful of bitwise operations.

As a whole, the VDB data-structure and OpenVDB implementation remain an extremely robust and extensive utility for all manners of volumetric use-cases. It's become ubiquitous in the computer graphics industry, and been used extensively for simulation PIC-like and otherwise. As the feature-set of OpenVDB continues to grow to this day, it is likely that the VDB data-structure will remain relevant to many industries and areas of study.

Despite its ubiquity, OpenVDB is not without its alternatives. In particular, Sparse Paged Grid (SPgrid), as introduced by Setaluri et al in 2014[46], provides a sparse grid which looses the flexibility of VDB in favor of simpler implementation and improved access speeds. Our GPU sparse grid contribution, VSPgrid, is a direct descendant of the Sparse Paged Grid, developing upon the original design as well as prior GPU adaptations.

SPgrid is a tree-less data-structure whose key insight is to create sparsity by leveraging the existing sparsity of modern operating system memory management. To utilize memory resources effectively, operating systems allocate physical memory as fixed size memory pages scattered about physical memory and alias sub-ranges of the virtual address space into these pages. This process of allocating and aliasing is done automatically by the operating system while user processes run, and is accelerated by CPU hardware. Operating systems also commonly expose this functionality as a utility for applications, allowing them to create separate virtual address spaces to be managed automatically by the OS.

It is this utility that SPgrid leverages. Each SPgrid is logically equivalent to a dense grid of equal size, however instead of allocating this full dense size, a virtual address space of that size is created. The address space behaves semantically like an impossibly large contiguous array of grid values, however behind the scenes only sub-ranges of the address space which are directly accessed will have memory pages allocated and bound. This allows users of an SPgrid to interact with the grid without concern for memory management or hierarchy traversal. All accessed grid locations will automatically become valid when first touched.

Despite using no hierarchical data-structure, SPgrid implicitly organizes its cells within a hierarchy as part of a spatially coherent addressing scheme. This scheme's primary purpose is to optimize the layout of values within memory. SPgrid defines

blocks to be collections of grid cells whose total size is equal to that of a single memory page. These blocks follow a space filling Z-order curve within the grid, meaning that there is a high probability that grid cells which are spatially adjacent will be nearby in memory as well.

Z-order curves are recursively defined, and refine a static spatial hierarchy with each iteration. Z-order curves arise naturally when an N-dimensional (always 3-dimensional for SPgrid) index space coordinate has its component value's bits interleaved. This interleaving, known as Morton encoding, can be accomplished through a small number of bitwise operations and results in a unique integral value. Similar to the hash/index values used by VDB, these encoded grid coordinates allow for rapid mapping of 3D coordinates to memory addresses or indices. Additionally, Morton encoded addresses implicitly locate grid blocks within the Z-order curve hierarchy. This scheme does however force SPgrids to exist within a limited bounding box, and does not support the infinite bounds of VDB.

SPgrid's simplified design and interaction mechanics makes it a much lighter weight sparse grid implementation when compared to VDB. Furthermore its memory allocation being so intimately tied to the underlying function of CPU based computing allowed it to achieve greater sequential and stencil access speeds on contemporary CPU. Being competitive with VDB and simpler to implement overall, SPgrid has maintained relevance in the field of simulation alongside its counterpart. SPgrid has also received increased attention as a result of interest in GPU based simulation methods. In particular, recent work on GPU driven material point methods implements a GPU adapted version of SPgrid as the sparse grid of choice.

The research work of Ming Gao promotes the use of SPgrid and GPU variants for implementation of the Material Point Method (MPM), which is one of the major

PIC-like methods[21]. He advocates for SPgrid on the merits of its implementation's simplicity, and the design's focus on applicability to high-performance simulation.

In their following work, *GPU Optimization of Material Point Methods*[22], Gao et al successfully adapt traditional SPgrid to the GPU under the name "GSPgrid". This work is the most closely related to the work of this thesis, both by providing the first GPU adaptation of SPgrid, and outlining a host of techniques for adapting MPM and PIC-like simulation methodology to GPU computing. Additional discussion of these details are provided in the following section.

The implementation of GSPgrid is a direct match for traditional SPgrid in several aspects. Both SPgrid and GSPgrid utilize the same spatially coherent addressing scheme and 4KiB memory page block alignment. However, unlike CPU SPgrid, the GPU platform does not offer the automatic virtual memory management offered by the operating system. GSPgrid's implementation must instead substitute itself in the role of the OS, manually allocating and aliasing memory pages for the virtual address space. GSPgrid achieves this by pre-allocating a large span of GPU memory to serve as a pool of memory pages. They then manually map all SPgrid addresses into this span using a manually created and maintained page offset table.

Although GSPgrid successfully recreates much of SPgrids function and virtues, it also looses some of SPgrids celebrated simplicity and dynamism. Management of grid memory pages falls upon the host application and must occur preemptively, instead of triggering automatically upon page fault. Additionally, the use of a user managed translation table likely comes with greater cost than the CPU's built-in translation look-aside buffer (TLB). Finally, GSPgrid does not exhibit the same sparse dynamism as SPgrid. GSPgrid pre-allocates all memory pages at initialization. The number of pages allocated is constant through the lifetime of the GSPgrid, and based on an estimate of the required active/inactive grid block ratio. If a greater portion of the

grid requires memory backing than expected, the GSPgrid will fail. In all other cases, the actual memory requirements of the GSPgrid will be lower than the estimate, and under-utilize the allocated memory.

Our VSPgrid implementation will address many of these weaknesses. We reintroduce complete memory dynamism to the GPU sparse paged grid, and utilize Vulkan's sparse memory utilities to provide GPU hardware with the opportunity to manage memory aliasing at a lower level. Additionally, while our memory management must also involve intervention by the host, we have simplified this process to minimize the complexity of GPU compute code's interactions with sparse grid memory.

3.3 Parallel and GPU PIC-like Methods

PIC-like simulation methods have frequently been brought into a parallel computing context in an effort to accelerate results. Prior works on this subject span multi-core parallelism, distributed computing parallelism, and the more recent adaptation of PIC-like methods to discrete GPU. Insights from these works overlap in the handling of memory layout, access patterns, domain decomposition, and data-race avoidance.

Ma et al[39] implement a parallel version of the Generalized Interpolation Material Point Method (GIMP) in 2D using an existing parallel simulation framework "SAMRAI". The authors believe that the results of sequential GIMP could be greatly improved if it was made both faster and more adaptive through parallelization. They demonstrate their hypothesis by implementing a parallelized GIMP simulation which focuses on handling nanoscale interactions between materials, a task which standard MPM could not handle due to the difference in detail between material interfaces and material bodies.

The SAMRAI framework specializes in decomposing simulation domains into rectangular grid cells, allowing adaptive refinement of the cells, and distribution of cells

to multiple processors. Ma et al harness SAMRAI by formulating a version of GIMP which will adaptively subdivide localized regions of the simulation domain according to their material contents. By detecting the interfaces between materials, they were able to adaptively subdivide the simulation until nanoscale interaction was possible.

The processing of each cell during a simulation frame is entirely independent, not only allowing hazard-less parallel processing, but allowing each cell to be processed with an adaptive timestep. Data-races at the interface between cells was resolved by duplicating boundary data between cells. These "ghost regions" between cells ensure each cell possesses all necessary data for its computations before processing begins. Duplication of ghost data was accomplished via inter-process communication.

The Uintah Software Package[23, 5], is a long lived software package for multi-processor scientific simulations. Uintah supports a theoretically unlimited diversity of simulation techniques, but includes existing implementations and tools for common simulation methods including GIMP. Uintah accomplishes generalized parallel simulation by defining language agnostic protocols for data-management, communication, and scientist manual intervention which culminates in an interactive scientific simulation experience. A master control process handles allocation of computing resources and scheduling, and utilizes MPI like communication to coordinate work between processors. A centralized "Data Warehouse" is used as an abstraction for simulation data-storage which prevents users from needing to resolve dependencies between components. The warehouse enforces single-assignment semantics and then optimizes internally to procedurally prevent the creation of data-races.

Chiang et al[13] and Dong et al[18] both implement GIMP on the GPU using NVIDIA's Compute Unified Device Architecture (CUDA) as their GPU programming platform. The authors note that a significant portion of computations within MPM exhibit data parallelism and are thus a good candidate for GPGPU compute. Both

implementations find the Particle To Grid stage of MPM to be a particular challenge due to the inevitable overlap of particle contributions to grid cells.

Chiang et al[13] tests two methods of handling the P2G stage. First they tried using GPU atomic addition operations to build a list of particles for each grid cell. It is then the cells themselves which are processed in parallel such that there is no data-race between GPU threads. As an alternative implementation, they sort particles based on their grid locations and use the sorted order to parallelize grid transfer without conflict. Comparing the two implementations, the authors observe better performance from the first but admit that they did not conduct a thorough investigation of how the two solutions might scale to larger simulations.

Dong et al[18] also use particle gathering to handle the mapping of particles to grid cells. Rather than deal with the challenge of conducting P2G transfer on the GPU, the authors transfer data back to the CPU and conduct P2G sequentially. They claim that the difference in performance with the GPU is trivial due to the small number of particles.

To the best of our knowledge, neither [13] nor [18] utilized any advanced data-structures in their implementations. This means that both implementations will inevitably be restricted to a relatively small dense grid resolution, and likely suffer from disk IO limitations when exporting results of their simulations.

Most relevant to this thesis, and one of the most contemporary GPU applications of PIC-like simulation is that of Gao et al[22]. Mentioned previously in the discussion of sparse grids, Gao's paper on GPU methods for MPM presents a well engineered GPU adaptation of MPM, making several key contributions. It is also the most direct ancestor of this thesis.

Like previous GPU MPM implementations, Gao's implementation is built on CUDA. The implementation is fully GPU driven, conducting all simulation opera-

tions on the GPU apart from GSPgrid page-table updates and export of simulated particle data. Along with their contributions to the computing methods for GPU MPM, the authors also demonstrate their methods with a heat solver and novel sand constitutive model.

As discussed in the previous section, the introduction of a GPU Sparse Paged Grid (GSPgrid) is a major contribution of the work[22]. The data-structure's sparsity allows for simulations to scale to high resolutions on the already memory constrained GPU hardware, while also being simple enough to minimize added overhead. The authors report that their GSPgrid out-performed prior implementations on both the CPU and GPU during the grid strenuous P2G stage.

The authors two other most significant computing contributions both relate to the handling of the P2G stage of MPM and other PIC-like methods. As described previously, the P2G transfer stage presents significant challenge for parallel PIC-like simulations due to grid cells commonly being subject to modification from multiple particles. Parallel PIC-like implementations typically handle data conflicts arising from P2G using either scattering or gathering.

Scattering techniques parallelize P2G on a per-particle basis, assigning particles to independent threads of execution which update grid cells in an unstructured manner. Scattering typically relies upon atomic operations to resolve write-conflicts. The gathering approach operates per-grid cell, by first building a list of nearby particles for each cell. Grid locations are then be processed concurrently, using their particle lists to accumulate their own value without conflict.

Gao et al[22] recognize that gathering is additionally difficult for GPU MPM due to the inability of executing GPU code to dynamically allocate its own memory. As a result, GPU gathering methods require pre-allocated memory and enforcement of limits on the total number of particles gathered within grid locations. Despite this

difficulty, scattering is also generally unappealing due to the cost of atomic operations at scale. The authors overcome this challenge with two algorithmic contributions: Particle Reordering and Intra-Warp Scattering[1]

Particle reordering refers to the author's partial sorting of particles per each frame of simulation. This sorting helps to maintain a mapping between grid locations for the sake of updating GSPgrid's page-table, and to keep physically adjacent particles near their neighbors in memory. The implicit spatial hierarchy of (G)SPgrid provides a convenient structure by which to organize particles into spatial buckets. Each particle's location can be converted into a spatial hashing key by virtue of the grid's addressing scheme. The authors sort particles by these keys using a GPU parallel histogram sort. This results in a particle array which is organized into contiguous spatial buckets trivially mappable to grid locations.

The grid locations mapped by the particle reordering are coarser than the resolution of the grid, instead gathering particles within larger blocks of grid cells. It is within these blocks where Intra-Warp Scattering is implemented. Gao et al introduce a novel local scattering method which utilizes CUDA warp intrinsics to minimize the cost of scattered writes to grid cells.

During P2G, each bucket of particles and associated grid cells are assigned to a CUDA warp. In general GPU computing terms, warps are a GPU computing unit consisting of many threads operating together as a unit to achieve SIMT execution. Threads within a compute unit share a limited set of resources, including a shared memory store. Some modern GPU offer support for specialized operations which coordinate data-sharing algorithms between threads in a compute unit. Exposed as

[1] "Intra-Warp Scattering" is not a term used by the authors. Their method is referred to simply as "Parallel Particle-To-Grid Scattering". I have renamed it here to make it distinct from generic parallel P2G methods.

warp-intrinsics in CUDA, they allow for efficient implementation of parallel reductions between compute unit threads.

Intra-Warp Scattering takes advantage of both warp local memory and intrinsics to optimize P2G transfer. Particle contributions to grid values are accumulated by threads within the warp. Warp intrinsics are then used to sum contributions between threads until a single representative thread holds the value to write into grid memory. Additionally, grid values are first written into a mirror of the local grid block stored in warp shared memory. Accesses to shared memory are typically much faster than those to global memory, limiting cost until the final copy of values to global grid memory occurs.

Despite its many merits, the GPU MPM implementation of Gao et al is not without its weaknesses. As described in the previous section, their implementation of GSPgrid lacks the true dynamism of SPgrid, and some of the original data-structure's simplicity. Beyond the limitations of GSPgrid, their implementation largely overlooks the impact that data-transfer and disk IO will have on the final GPU MPM application. Although their GPU simulation algorithms exhibit high parallel throughput, they must ultimately transfer each frame of particle data to the CPU and then to disk for final output. These transfers are handled naively, and thus they risk becoming bottlenecked by GPU to CPU transfers and disk output, loosing scalability for large volumes of particle data. This thesis will address both of these shortcomings in the development of our own PIC-like simulation framework.

3.4 GPGPU Computing in Vulkan

There is a very limited body of work on the application of the Vulkan API to GPGPU problems, and an even more limited body of work relating to GPU driven simulations. This is not totally surprising; Vulkan is a young API which is not advertised or

designed to be a direct competitor to GPGPU platforms such as CUDA or Khronos' own OpenCL. Despite this fact, Vulkan does present some unique opportunities for GPU driven computing and the engineering of high-performance GPU software. Additionally, current offerings of GPU API did not appear to unilaterally cover the requirements of this thesis at the time of its development. In particular, only Vulkan offered the sparse memory functionality we required for our improved GPU sparse paged grid, although since that time this functionality has been added to CUDA. Apart from technical considerations, we find cross-platform and open standards to be generally more appealing.

To the best of our knowledge there are currently almost no academically published works on conducting PIC-like simulation or related physically based simulations in the Vulkan API. The most relevant written work we could find is the thesis of Samuel Ivan Gunadi[25], which implements Smooth Particle Hydrodynamics (SPH) on the GPU using OpenGL and Vulkan. Smooth particle hydrodynamics is fully Lagrangian fluid simulation method making it a partial analog to our focus on PIC-like methods. However, their thesis is focused primarily on the implementation details of SPH, and not on the engineering of the Vulkan implementation. Comparisons with our own work is only superficial, relating common elements of Vulkan API usage such as writing SPIR-V compute shaders and applying synchronization primitives. The paper's only conclusion in regards to Vulkan driven simulation is that it outperforms an equivalent OpenGL implementation at scale.

Although lacking formal publications, there are a handful of open-sourced PIC-like simulation implementations based in Vulkan. Two which stand out are *Vortex2D*[40] by Maximilian Maldacker, and *Real-time Particle-based Snow Simulation with Vulkan*[12] by Qiaosen Chen and Haoyu Sui. The first is a self-described FLIP and PIC based 2D fluid simulator implemented entirely on the GPU using Vulkan

compute shaders. The second implements an MPM based snow simulator implemented in both CUDA and Vulkan.

Vortex2D's documentation is geared towards documenting usage of the implementation for running simulations rather than explaining the inner workings. However from what is available, we know that this implementation uses GPU textures as a dense computation grid. Beyond what is written, a cursory inspection of the project's source code suggests at least one notable similarity with this thesis, the use of a parallel prefix scan and particle bucketing for P2G transfer.

The snow solver implemented by Qiaosen Chen and Haoyu Sui provides moderately more written details than Vortex2D. Their work is an MPM implementation based primarily on Goswami et al[24] and Stomakhin et al[47]. Their implementation manages to achieve real-time 3D snow simulation using GPU compute and the methods of their related works. Again, implementation details are scarce and few conclusions can be made without an in-depth review of the projects source code and reproduction of their results. We can however tell that this implementation too uses a uniform dense grid for Eulerian computations and is thus limited in scale.

Chapter 4

IMPLEMENTATION OVERVIEW

This thesis' summary contribution is a high-performance GPU framework for particle-in-cell type continuum simulations. We build this framework using the Vulkan® API which we govern from a C++17 host application. The application takes advantage of Vulkan's support for host parallelism to make the primary simulation loop extensively asynchronous on both the CPU and GPU. This asynchrony improves throughput by providing the GPU with a more consistent workload and mitigating the bottlenecks of memory transfers between the CPU, GPU, and disk.

Documentation of our implementation will be structured as follows. Section 4.1 will discuss high-level architecture, describing the basic control flow of the framework and highlighting some key aspects of its design. Section 4.2 will explore some nuanced details of the host program implementation not sufficiently covered by other sections. Chapters 5 and 6 are dedicated to the details of the GPU sparse paged grid and particle resources provided by our framework. Finally, Chapter 7 will bring together all the prior chapters to discuss the architecture of the primary simulation loop.

4.1 PIC Simulator Framework Architecture

The top-level control flow of our host application is simple. We begin by initializing the Vulkan API and configuring the physical GPU which will be used to drive simulation. We then setup a variety of Vulkan resources which are necessary regardless of the specific simulation configuration. After this outer initialization step, the host application begins iteration over a collection of PIC pipeline specifications.

Each specification provides instructions for the configuration of the framework, including a sequence of GPU compute code modules containing the simulation logic. The specifications are consumed one at a time and built into an internal representation. Once built, the pipeline can be executed by the framework's main simulation loop.

Currently pipeline specifications come in the form of JSON objects with a structure specific to our framework. They are passed into the host application as file paths on the command-line. At present, our host application does not have any form of GUI. Simulation results are written to disk to be viewed in external software.

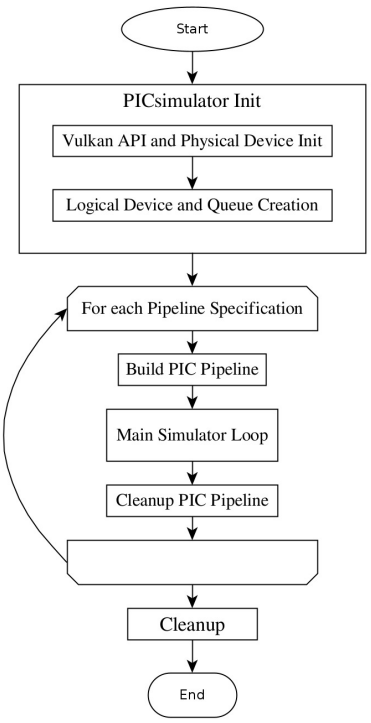


Figure 4.1: Top-Level Control Flow

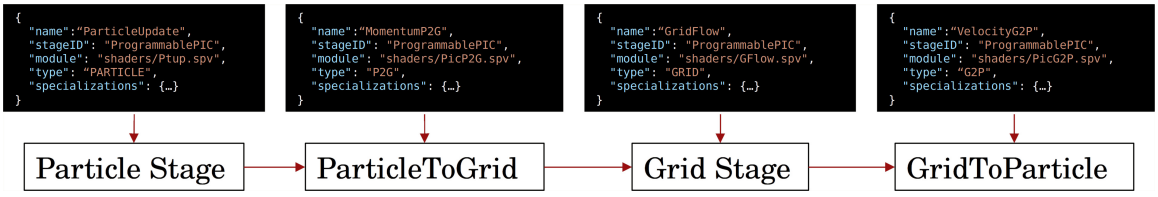


Figure 4.2: Example GPU compute modules within JSON PIC pipeline specification.

The main simulation loop runs once for each specification provided. After the loop has completed execution, all resources associated with that specific pipeline specification are released. When all provided pipeline specifications have been executed, the application tears down the remaining objects and terminates.

4.1.1 Programmable PIC Pipeline Overview

The programmable pipeline of GPU compute stages is at the center of our framework. An arbitrary number of stages are supported, each capable of being granted access to both Lagrangian and Eulerian resources. Each stage is defined by a SPIR-V GPU compute code module, a type identifier, and optional specialization constants. A stage's type identifier indicates which resources Eulerian, Lagrangian, or both, it will be accessing. Beyond indicating which resources to bind, stage typing information is used by the host to automatically structure resource access synchronization.

Specialization constants are a feature of SPIR-V and Vulkan which allow low-cost run-time modifications to GPU code. When SPIR-V IR is passed to a Vulkan implementation it is translated into a native format for execution on the hardware. Specialization constants are run-time values passed along with the IR code, which then become compile-time constants as the IR generates native code. This affords high-level GPU code an increased level of flexibility with little to no performance cost as constant value optimizations may be applied during SPIR-V IR translation.

Specialization constants provided for stages in a PIC pipeline specification allow PIC compute modules this flexibility in configuration, and may allow code-modifications to be forgone in favor of specialization constant tweaks. Specialization constants also play a critical role in our sparse grid implementation as will be described in Chapter 5.

4.1.2 Simulation Resource GLSL Header Interface

It was a challenge to provide our programmable PIC pipeline with the flexibility to support a range of potential compute activities without overburdening each compute module with the complexities of resource management. Management of our frame-

work's resources, the sparse grid especially, though not computationally expensive, does require a large quantity of verbose GPU code. The complexity of this code in league with a lack of both static and dynamic safe-guards in GLSL and the absence of a semantic link between C++ host code and GLSL code, make it exceedingly easy to introduce programmer error. To minimize this risk, we provide a simplified functional interface for our PIC resources.

Compared to most other high-level languages, GLSL is lacking in code-reuse and interoperability features on both a language and tool-set level. This is understandable from a historical perspective, but means that the language may buckle under the weight of larger compute applications for which it was not originally designed.

The rise of Vulkan® and SPIR-V has allowed for GLSL to be compiled externally by a common and implementation agnostic compiler[4]. Among the *Shaderc* project's many valuable utilities is its addition of C-style `#include` statements through an unofficial GLSL language extension. We use this functionality to define a common interface for our PIC simulation framework with GLSL header files.

The interface is separated into three files: "simulation.glslh", "particles.glslh", and "vspgrid.glslh". The first two are minimal, providing general simulation context such as the current frame number, timestep, and exposing the Lagrangian particle buffers. "vspgrid.glslh" is considerably larger and more involved. It exposes all of the buffers used by the VSPgrid data-structure, but more importantly, defines many functions for interfacing with the structure. The majority of these functions relate to the translation of grid coordinates between world space, index space, and packed grid addresses. This will be elaborated upon in Chapter 5.

Using GLSL header files is effective, but unfortunately crude. Header file inclusion is implemented by direct inlining and carries no notions of scope or ownership. As a result, all buffers and utilities must be exposed publicly even if not intended for

direct use. It is similarly not possible to organize utilities using namespaces, objects, or other similar tools through GLSL. Since headers have no contextual awareness and GLSL lacks meta-programming, headers are not able to procedurally determine the correct resource binding locations in all cases either. As a result, code using the interface may be required include headers in specific order and/or manually override resource binding locations.

GLSL also lacks type parametric polymorphism making it difficult to support arbitrary data-types for Lagrangian particles and Eulerian grid cells. Our solution is to represent each Lagrangian particle and Eulerian cell with a GLSL structure. The definition of the structure can be overridden through the pre-processor. It is important to note however that altering this structure does not inform the host program of any resulting change in size requirements or alignment. Creating a formal and automatic connection between host and GPU data-types is a goal for future work.

4.2 Host Application Details

4.2.1 Logical Device and Queue Setup

As stated earlier, our application begins by initializing the Vulkan API as well as the physical device which will drive the simulation. This portion of the code is fairly boilerplate as Vulkan applications go. Initialization of the logical device is however more involved. Our framework requires that we enable several optional Vulkan features which are enumerated in Appendix B. We must also carefully select an optimal queue configuration to take full advantage of available hardware.

The goal of our queue configuration selection is to maximize the potential for asynchronous GPU workload execution. The latest generations of discrete GPUs provide support for asynchronous GPU workloads[28]. In Vulkan asynchronous workloads

are typically exposed through different 'queue families'. Although not guaranteed by the Vulkan specification, it is expected that current discrete GPU hardware offering multiple queue families may support some level of asynchrony between queues of each family[27]. Importantly it is not expected, but similarly not disallowed, that different queues from the same family will operate asynchronously. With these assumptions in mind, our implementation attempts to draw queues from as many distinct queue families as possible.

Our implementation always creates three `VkQueue` objects, each with a specific purpose. The Vulkan API guarantees that any device supporting graphics operations must supply at least one 'queue family' which supports the three 'core' operation types: graphics, compute, and transfer. The first of our three queues is a core queue created from such a family. Since it is the most flexible, the core queue is used for all operations outside of the main simulation loop, and for miscellaneous operations during the simulation.

The remaining two queues are a dedicated compute and a dedicated sparse binding/transfer queue. Whenever possible, these queues are drawn from distinct queue families such that, under ideal circumstances, all three queues might participate in asynchronous operation. This three queue arrangement is designed to match up with current consumer GPUs which typically offer one core family, one dedicated sparse binding family, and one dedicated compute family[28, 27, 53].

In the event that the hardware does not provide the additional queue families, we fallback to the core family. Our implementation is written such that no code depends on the number of unique queue families. When a GPU cannot support asynchronous queues, our framework will still operate as expected, but without the benefits of GPU asynchronicity.

4.2.2 Miscellaneous Vulkan Resource Setup

The particles and sparse grid represent the bulk of complexity as far as resource management is concerned. However, there are several general considerations to be made when managing Vulkan resources within the entirety of a Vulkan host application.

In Vulkan, 'descriptors' are an object used to reference resources such as buffers or images. They are allocated from 'descriptor pools' and gathered together as 'descriptor sets'. One or more of these descriptor sets can be bound to Vulkan graphics or compute pipelines prior to execution to make those resources available to the executed GPU code.

We allocate all descriptors for our framework's resources using a single descriptor pool as the number needed is static. We then group the descriptors into one of three sets: the simulation set, particles set, and grid set. As their names would suggest, they hold the descriptors for all GPU buffers associated with general simulation information, Lagrangian particles, and the sparse grid respectively.

The particle and sparse grid descriptor sets contain multiple descriptors each and their description will be provided in Chapters 5 and 6. The simulation descriptor set is however simple, containing a single uniform buffer. This uniform buffer is host coherent and mapped to a struct which is mirrored between C++ host and GLSL GPU code. The struct contains global simulation variables such as the current simulation frame, simulation timestep, and global simulation time. Modifications to these variables are made on the host and automatically mirrored on the GPU.

Designating simulation resources to these separate descriptor sets allows for GPU compute modules to be bound only to the subset of resources they require. This makes the scope of compute module resource usage more concrete and allows GPU compute code to safely exclude definitions for unused resources.

Chapter 5

VULKAN SPARSE PAGED GRID

Although PIC-like simulations are more Lagrangian than Eulerian in principle, the use of an Eulerian background grid characterizes PIC-like methods in practice. The impracticality of densely allocated grids at scale has resulted in the invention and application of many alternative grid data-structures. We implement a GPU local sparse grid by adapting the design of the *Sparse Paged Grid* conceived by Setaluri et al [46] as an alternative to contemporary tree based sparse grids.

GPU programming lacks dynamic memory allocation, and generally benefits when branching control flow is avoided. This makes the SPgrid's tree-less design ideal for use on the GPU. SPgrid has previously been adapted to the GPU as part of the GPU MPM implementation of Gao et al [22] in 2018. This thesis' implementation shares many traits with Gao's GPU SPgrid by virtue of their common ancestor. However, our implementation differs crucially in regards to memory management and GPU code interface.

The GPU MPM solution implemented in Gao et al was written in CUDA®. At the time the work was completed, CUDA did not possess any of the virtual memory capabilities which characterize CPU SPgrid. Their solution was to emulate a virtual memory space by manually re-mapping SPgrid addresses into a large `cudaMalloc()` allocation using a separate table of offsets. The size of the allocation is constant, and set using a manually estimated upper-bound on the percentage of the grid which will be utilized.

This method is effective in avoiding the cost of a dense grid, but does not dynamically adapt to the needs of the simulation. If the estimated upper-bound on grid

occupancy is exceeded, the simulation will fail or require complete re-initialization of the grid. If the upper-bound is never approached, the application may be consuming far more memory than is ever used. Furthermore, the use of a separate page offset table adds additional logical complexity to GPU code utilizing the sparse grid.

This thesis' implementation of a Vulkan GPU sparse paged grid does not exhibit these restrictions. Our GPU sparse paged grid, henceforth abbreviated 'VSPgrid', leverages Vulkan's support for virtual memory spaces to create a sparse paged grid whose design is closer to that of CPU SPgrid, and which exhibits complete dynamism in its memory footprint.

5.1 Overview

Vulkan sparse buffers and sparse residency features are the pivotal mechanism behind our VSPgrid implementation. A sparsely resident buffer appears to GPU code as if it where any other contiguous span of memory. In reality, each sparse buffer is representative of a virtual address space mapped non-contiguously with GPU memory allocations. In effect this utility is very similar to the type of virtual address space utilities upon which the original SPgrid was built. It retains the simplicity of direct access from the GPU code perspective, and allows for the grid's memory footprint to scale in congruence with demand.

Just like Setaluri et al's original SPgrid, we map the index space of a 3D dense grid to 1-dimensional offsets into the virtual address space using a spatially coherent addressing scheme. This addressing subdivides the grid domain into 'blocks' which each contain a quantity of data equal to the size of a single memory page. The sparse buffer is aligned to these blocks such that there is a 1-to-1 mapping between blocks and pages of memory. The spatial coherence of the block ordering provides benefits of locality both with respect to memory cache and allocation of memory pages.

Vulkan sparsely resident buffers however do significantly differ from SPgrid's usage of `mmap(..)` in regards to how the virtual address space is populated with memory pages. Part of the ease with which CPU SPgrid can be managed is due to the implicit management of their virtual address space by the operating system. The CPU and operating system extensively support virtual memory and memory paging as they are critical to normal system operation. Likewise the virtual address space used by SPgrid can be accessed randomly with the expectation that the OS will handle all page faults without intervention by the application.

Vulkan sparse buffers are not supported by any such page fault handling mechanism. Instead, as with other Vulkan buffers, memory must be explicitly managed by the host application. Applications must allocate individual memory pages, then bind them to sub-ranges of the sparse buffer using `VkQueueBindSparse` submissions. Since none of these activities can be conducted from GPU code, the host application must either fully predict the pattern of access into the sparse buffer or maintain a dialogue with running GPU code.

Our VSPgrid implementation uses the later method, and many of the mechanisms which makeup the grid implementation are dedicated to maintaining this dialogue. We wanted our VSPgrid implementation to favor minimization of GPU code complexity whenever possible, and so we proceeded to create our sparse grid such that the details of memory management are almost completely hidden from GPU code. Our design allows GPU code to worry only about grid data addresses, which are just indices into the dense grid.

Since we cannot interrupt and resume GPU code to handle page requests, it is not possible to enable fully on-demand random access the way CPU SPgrid does. Instead, we make grid usage a two-step process of 'requesting' then 'accessing'. The request step determines the random-access grid addresses it wishes to utilize. The

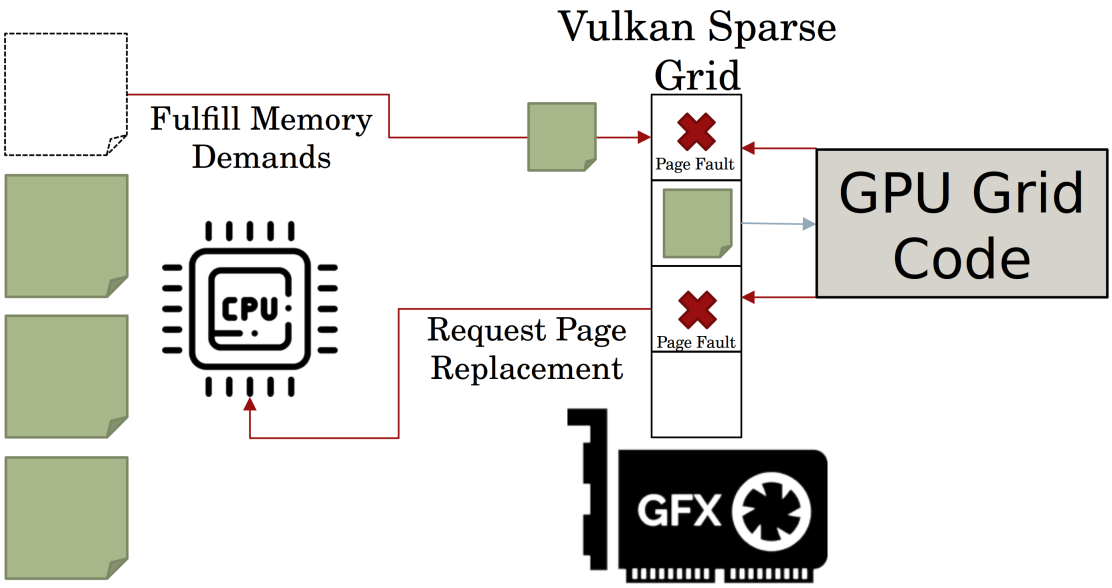


Figure 5.1: VSPgrid page request then access diagram.

addresses are then passed to a 'request' function which will flag the appropriate grid regions for memory backing. Once requests have been made, the host is signaled to handle the GPU's demands for memory. Upon resolution of grid requests, the second phase of GPU code is executed and can freely access all requested grid addresses. The interaction is simple from the GPU programming perspective, and the cost of memory management can frequently be hidden by requesting memory early and deferring grid computations in favor of non-grid GPU computations which are executed concurrently with the host's grid management routine.

5.2 VSPgrid Configuration and Vulkan Resources

Within the host application, the VSPgrid is represented by a monolithic VSPgrid class which handles all creation, management, and teardown of grid resources. Each instance of the class begins construction with specification of a world-space bounding box defining the grid's domain, a triplet of integers defining the resolution of the grid, and the size (in bytes) of each cell within the grid. Starting with these three variables,

the VSPgrid computes a lengthy collection of constants describing the grid's configuration. This configuration is necessary for both setup and continued maintenance of a VSPgrid.

At the root of most of these extrapolated constants, is the relationship between the data-size of each grid cell, and the memory page size of the Vulkan implementation. VSPgrid, like SPgrid, is designed to organize grid cells into larger cuboid blocks, such that each block's size in memory is equal to the size of a memory page. This packing of cells into blocks creates a consistent geometric relationship between regions in the 3D and the memory pages used to back the sparse grid.

Much like OS memory pages which are typically 4KiB in size, Vulkan sparse buffers must be bound only to memory pages of predetermined size and alignment. This minimum size and alignment is defined by the Vulkan implementation, and so the VSPgrid class must derive valid block dimensions dynamically. The derivation of most other constants in the VSPgrid configuration require knowledge of block dimensions as well.

To compute appropriate block dimensions, the Vulkan sparse buffer memory page size is retrieved from the implementation. The page size is then divided by the quantity of memory required by each grid cell. The quotient is then rounded down to the nearest power of two forming the 'block volume'. The block volume is then factored into three smaller powers of two which form the block dimensions. Block dimensions are not required to be equilateral. Once block dimensions are calculated all other grid measurements can then be derived and adjusted to meet the requested grid dimensions. These constants are also used to derive the appropriate addressing scheme for the grid, and generate the necessary bit-masks.

Each instance of our Vulkan sparse grid implementation is formed from four separate Vulkan resources. Most important is the device local sparse buffer which acts as

the grid's virtual address space for cell data storage. The other three buffers are all host coherent buffers used to communicate the state of the sparse grid. The sparse buffer is initialized to a capacity matching the dense equivalent of the grid configuration, but no physical memory is allocated at initialization. The buffer is accessed from GPU code using a bindless 64-bit physical storage buffer address rather than descriptor bindings to enable more pointer-like access semantics within GPU code.

Bitmaps are the primary data-structure for tracking and updating the topology of the memory backed blocks within the grid. Two bitmaps are maintained over the lifetime of each VSPgrid, each with length equal to the total number of blocks contained by the grid's address space. The first bitmap is read-only within GPU code and indicates the current state of the grid. In both bitmaps, each bit represents a block within the grid. A block is considered 'active' when it is backed by a memory page, and this state is concisely indicated by a corresponding "1" in the bitmap. The second bitmap follows the same convention as the first, but is writeable from GPU code. GPU code may flip bits within the writeable bitmap to indicate requests for a change in grid memory topology.

Each VSPgrid uses one last host coherent buffer which does not play an active role in grid management, but does act as a utility for writing GPU code which accesses all active grid regions. Every time the VSPgrid memory manager fulfills grid memory requests, it logs the full list of active grid blocks into a read-only buffer. This list of active blocks is exposed to GPU code as a simple variable length array of block indices. GPU code can then use straightforward parallel iteration over the list to access all active regions of the grid without needing to scan the entire bitmap for active blocks. Many grid operations, for PIC simulation or otherwise, such as stencil operations, benefit from using the described coding pattern.

The GPU code interface to the data-structure provided by "vspgrid.glslh" contains both resource bindings and utility functions for intuitive grid access. The grid configuration constants computed by the CPU at initialization are necessary for reasoning about the grid's geometry and addressing scheme and are therefore necessary within GPU code as well. Since the configuration is constant for the lifetime of each VSPgrid, they are passed to GPU code as specialization constants.

Because specialization constants become compile-time constants, our VSPgrid addressing computations and other configuration dependent operations compile down to a nominal number of arithmetic operations. Specialization constants are a limited resource, so only a minimal subset of the host side constants are passed. The remaining configuration constants are recalculated within GLSL constant expressions which should evaluate and inline at compile time resulting in no additional consumption of specialization constants or registers.

5.3 Addressing Scheme and Memory Layout

Central to the implementation of any sparse paged grid is an addressing scheme which maps 3D grid locations into the 1D virtual address space. The details of the addressing scheme devised can have significant impacts on performance, and wise implementation choices allow the addressing scheme to provide additional utilities to sparse paged grid users.

Both SPgrid and VSPgrid devise highly similar addressing schemes which translate 3D grid coordinates into long (up to 64-bit) unsigned integer offsets into the virtual address space. Rather than use naïve lexicographic ordering of grid blocks, original SPgrid and VSPgrid both use Morton encoding to arrange grid blocks along a space filling Z-order curve. This makes the mapping of grid space to memory pages spatially coherent, providing benefits of locality. However Morton encoding, decoding, and

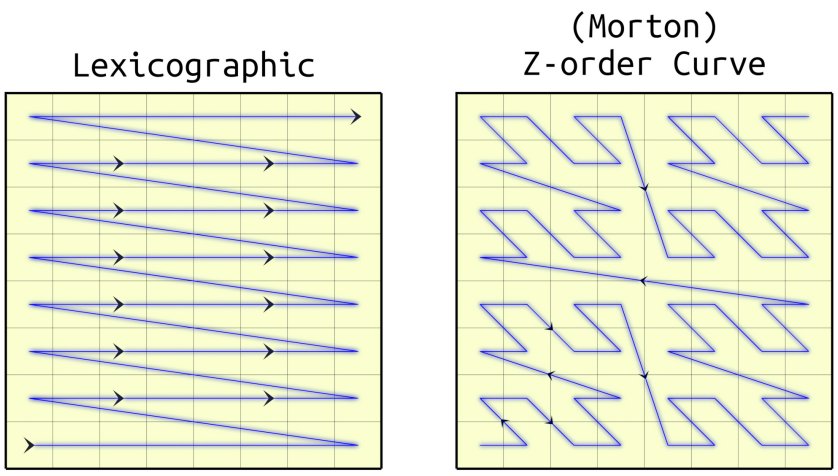


Figure 5.2: Side-by-side comparison of lexicographic ordering compared to Z-Ordering (Morton encoding).

arithmetic are slightly more costly than the lexicographic alternative. As a result, Morton encoding is only used at the block granularity. The cells contained within each block are themselves ordered lexicographically. Using this mixture of Morton and Lexicographic encoding balances the benefits of memory adjacency with the overhead of Morton encoding.

Our description thus far is a very high-level overview of the commonalities between SPgrid and VSPgrid addressing schemes, but leaves out critical details. The prior description also intentionally avoids discussion of the many significant differences in our addressing scheme as compared to original SPgrid. Certain aspects of our addressing make sense only when contextualized relative to SPgrid, but we wish to avoid providing a lengthy description of both implementations. To compromise we have listed a summary of the differences between SPgrid and VSPgrid addressing below. Their meaning and significance should become more clear after the reader has absorbed the details of our implementation, afterwards a more detailed breakdown of the differences and their justifications will be given.

1. Where as SPgrid individually addresses all bytes in grid memory, our scheme is an indexing scheme which covers all grid cells.

2. SPgrid's addressing implies a Structure of Arrays (SoA) layout, while we assume an Array of Structures (AoS) layout.

3. We introduce 'sub-blocks' which follow Z-curve order like SPgrid blocks but are smaller than the memory page size.

5.3.1 Addressing Scheme Implementation Details

As described, the VSPgrid addressing scheme uses both Morton and lexicographic encoding of 3D grid coordinates to form 64-bit address values[1]. The least significant bits of each address lexicographically encodes the location of a grid cell relative to the sub-block within which it resides. Conversely, the most significant bits of each address encode the location of the containing sub-block using a Morton encoded 3D coordinate.

The term 'sub-block' deviates from the terminology introduced thus far, but is a simple extension of the 'blocks' discussed previously. In our VSPgrid implementation, the term block is reserved to refer to a collection of cells or sub-blocks whose size aligns with the memory page size of the Vulkan sparse buffer. In all contexts besides VSPgrid memory management, sub-blocks are used in the exact same way as blocks are used in SPgrid. Just like blocks, sub-blocks group cells into cuboid collections, are placed along a Z-order curve, and store their contained cells in lexicographic order. They are

[1]GLSL compiler versions available at the time of this paper's writing support 64-bit unsigned integers through extension, but do not support using 64-bit integers in array subscript expressions as the language specification permits only `int` and `uint` primitives [35]. Despite this, once compiled down to SPIR-V, use of 64-bit values in array accesses are well defined and supported by our target platforms. To workaround this obstacle we modified the GLSL compiler to permit use of generic width integer types within array access expressions. We hope that in the future this functionality will be included in release versions of the GLSL compiler when targeting SPIR-V. Alternatively, compute code could be written directly in SPIR-V or another shader language, and 32-bits is sufficient for most grids less than 2048^3 in resolution.

defined identically to SPgrid blocks except for the fact that they are smaller than the memory page size. Geometrically, they can be viewed simply as octal subdivisions of the larger page aligned grid blocks. The justification for introducing sub-blocks is given in Section 5.3.2.

To compute the address for a grid cell, we begin with the intuitive 3D index space coordinates for the cell: (i, j, k). Index space coordinates are 3D integer vectors which span the range between $(0, 0, 0)$ and the full grid dimensions $(\mathbf{G}_x, \mathbf{G}_y, \mathbf{G}_z)$. The global cell index space coordinate is first split into a 'sub-block coordinate' $(\mathrm{SB}x, \mathrm{SB}y, \mathrm{SB}z)$ and 'local cell coordinate' $(\mathrm{C}x, \mathrm{C}y, \mathrm{C}z)$ by taking the integer quotient and remainder of the global cell coordinate divided by the dimensions of a sub-block.

Next both the sub-block and local cell coordinates are encoded as unsigned integers. The cell local coordinate is encoded lexicographically, which is easily accomplished by concatenating the bits of the individual x,y,z components of the coordinate. Encoding the sub-block coordinate is a more involved process. While lexicographic encoding is a concatenation of component bits, Morton encoding is an interleaving of component bits. The least significant bits of the coordinate components become the three least significant bits of the encoded values. Those three least significant bits are then followed by the second least significant bits of the component values, and so on until all non-zero bits are consumed. An example follows:

$$(x, y, z) = (x_2x_1x_0,\ y_2y_1y_0,\ z_2z_1z_0), \quad \text{mortonEncode}((x, y, z)) = z_2y_2x_2z_1y_1x_1z_0y_0x_0$$

$$\texttt{(3, 6, 4) = (011, 110, 100)}, \quad \texttt{mortonEncode((3, 6, 4)) = 110011001}$$

The most intuitive method for implementing this type of bit interleaving uses a loop and repeated bit-shifting. However this method is generally known to be non-optimal and it comes with additional cost in the GPU compute context where divergent control flow can create inactive threads. Fortunately, a totally branchless algorithm exists which can interleave bits using only magic numbers and bitwise

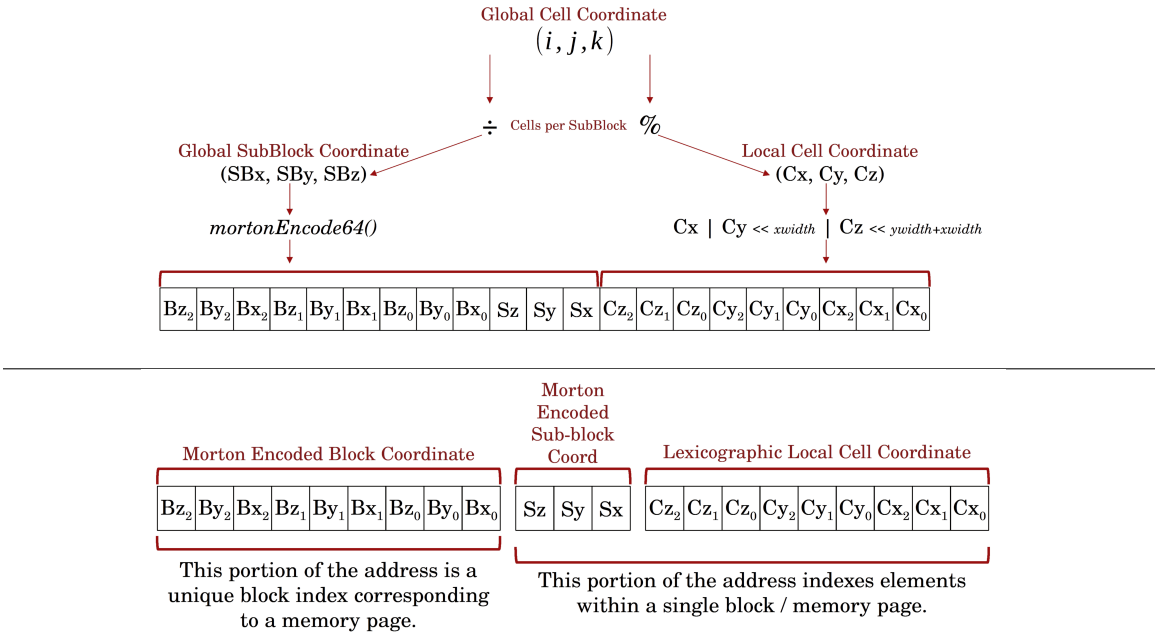


Figure 5.3: Addressing Scheme Bit Layout and Computation

operations. Our implementation of both Morton encoding and decoding is based on the magic number implementation of 2D bit interleaving published by Sean Eron Anderson[8]. We extended the algorithm to 3D and increased the scope so that it can encode values with up to 64-bits. This algorithm, written in GLSL and compiled into SPIR-V, results in 19 bitwise operations per encoding. The precise cost in cycles is dependent on the GPU hardware, but given that the function is purely arithmetic and data-parallel, performance should be near optimal. The GLSL source for these functions are provided in appendix A.

Once both the sub-block and cell local coordinates are condensed into their respective encodings, the final address is formed by concatenating the two encodings in to a single long unsigned integer. This process and the final layout of bits is illustrated in Figure 5.3. Note that the encoded sub-block coordinate by definition includes the coordinate of its parent block within its more significant bits. As a result identifying the memory page location for grid cells is trivial once their address is computed.

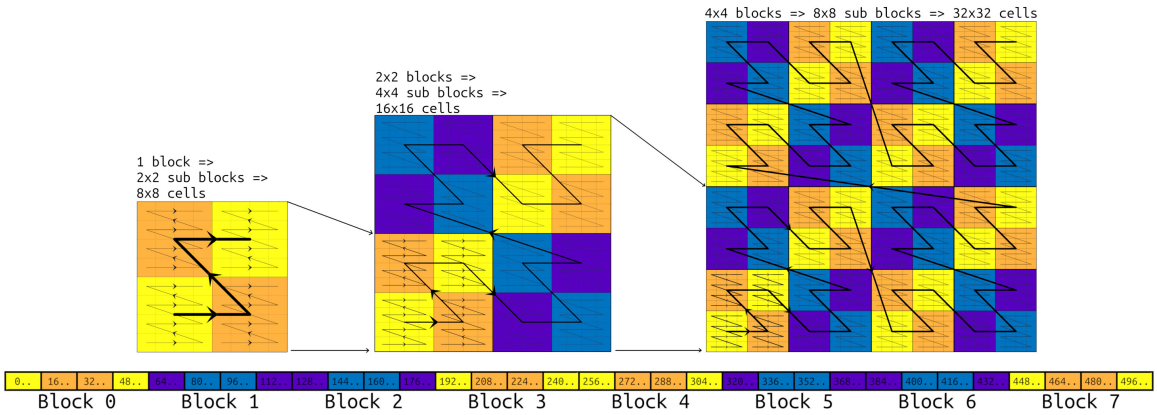


Figure 5.4: Demonstration of VSPgrid's addressing scheme in 2D. A high DPI copy and display is recommended.

It is important to not only understand the computation and bitwise layout of VSPgrid addresses, but also to understand how the addressing scheme organizes 3D grid space into memory. Figure 5.4 provides a 2-dimensional equivalent of the VSPgrid addressing scheme applied to a 32x32 grid.

Moving left to right, the figure initially shows a single grid block which contains 4 sub-blocks of 16 cells each. The sub-blocks follow Z-order, while the cells within are ordered row by row (lexicographic). In the middle, a set of 4 VSPgrid blocks is shown, each with their own set of sub-blocks. Observe that the Z-order pattern continues recursively, traversing each block in the same Z-pattern as it does with the contained sub-blocks.

Finally on the right hand side, we see the a full view of the grid where the Morton encoding creates yet another recursion of the Z-order curve. The checkerboard color patterns are a visual aid to distinguish blocks and sub-block boundaries, and do not carry any other significance. Finally, at the bottom of the figure we see how the first 8 blocks of grid memory would be mapped in relation to the example given. Each element in this bottom array stands in for a sub-block in the grid.

5.3.2 Deviations from SPgrid Elaborated

Now that the addressing scheme is fully specified, we can circle back to the list of deviations between our VSPgrid addressing implementation and the original SPgrid addressing scheme.

(1). The original SPgrid data-structure designs its addressing scheme to have C-like pointer semantics. Regardless of grid data type, an address into an SPgrid uniquely identifies a byte of memory within the virtual address space. We write our GPU compute code in GLSL, a language with no pointer types. As a result, our VSPgrid 'addresses' are not addresses in byte-addressable memory, but indices into a typed C-like array. We feel that this change comes at no cost of functionality as grid data can still be interpreted using whatever type the programmer wishes, including 8-bit types.

(2). SPgrid's addressing scheme also differs from our own as a result of their use of a Structure of Arrays layout for grid data. 'Structure of Arrays' (SoA) and 'Array of Structures' (AoS) are two opposing conventions for the layout of memory when storing values en masse with multiple data-types. Using grid data as an example, an AoS implementation such as ours defines grid variables as members of a structure. The entire grid is represented an array of these structures. In the SoA layout, as is used in SPgrid, data is instead organized into channels. Each grid variable is assigned to its own channel, which stores its values in a contiguous array. The entire grid is then represented as the collection of channel arrays, rather than a single array. This is illustrated by the GLSL code example in Figure 5.5.

Depending on context, a PIC-like simulation implementation may prefer AoS or SoA for reasons of performance or convenience. The original SPgrid uses SoA layout, and thus a portion of the addressing scheme is used for channel offsets. As discussed, we choose to use an addressing scheme that indexed into a typed array instead of

```
//AoS convention
struct GridCell{
    vec3 velocity;
    float mass;
    float temperature;
} gridData[NUMCELLS];
```

```
//SoA convention
struct GridDataChannels{
    vec3 velocity[NUMCELLS];
    float mass[NUMCELLS];
    float temperature[NUMCELLS];
} gridDataChannels;
```

Figure 5.5: GLSL Array of Structures VS Structure of Arrays example.

using memory offsets. As a result AoS is the natural layout for our implementation. It should be noted however that the choice of layout impacts only GPU code utility functions for translating 3D cell coordinates into array indices. An SoA alternative could be formulated without requiring any changes to the underlying operation of VSPgrid. Providing such a utility is a goal for future work.

(3). SPgrid's addressing scheme is predicated on the knowledge that operating system memory pages will be 4KiB in size. Vulkan sparse buffers do not offer any such guarantees and indeed the standard page size is 64KiB on our hardware and other dGPU we have polled. Although a larger page does not fundamentally change the way memory is managed, it does change the geometric scale at which cell addresses are spatially coherent.

Learning from the GPU MPM work of Gao et al [22], we know that the Morton encoding order of blocks provides, in addition to its performance benefits, a critical tool for spatial binning and grid domain decomposition for the purposes of parallelization. 4KiB of grid data is a quantity well suited for GPU parallel computation, and fits nicely into workgroup shared memory. 64KiB is significantly larger, and will just barely into GPU compute shared memory on recent generation AMD discrete GPU, and could not fit onto any other modern hardware[53]. Even assuming shared memory is not critical to a simulation's GPU code, using 64KiB grid blocks as GPU parallel work-items may be sub-optimal depending on the type of work being done and number of active blocks.

To overcome this challenge and implementation uncertainty, we introduce sub-blocks into our addressing scheme. Where as SPgrid uses Morton encoding to order only page aligned blocks, we include additional iterations of Morton encoding to subdivided memory page blocks into smaller units. By continuing to use Morton encoding to order blocks, we maintain spatial coherence at a finer granularity. The VSPgrid memory manager can easily ignore sub-blocks by right shifting away the least significant bits of the encoded address.

5.4 Grid Memory Management and Page Replacement

As previously introduced, unlike a typical OS backed virtual address space, Vulkan sparse buffers must be managed by the host application. Additionally, since GPU compute code cannot pause and resume to allow page replacement to run, code using VSPgrid must use a two step process of requesting grid data then accessing grid data. The result is a continuous dialogue between GPU and host in order to dynamically maintain the memory topology of the sparse paged grid.

This dialogue is primarily enabled by two host coherent bitmaps, where each bit uniquely identifies a block and indicates its state. One bitmap communicates which blocks of the grid are currently active, and the other bitmap gets modified by the GPU to indicate the requested future state of the grid. This method of communication is agnostic to any particular allocation or page replacement scheme, allowing the VSPgrid host to implement any variety of algorithms.

Presently we support two page-replacement modes: persistent and non-persistent. Non-persistent bindings are the default and typical of PIC-like simulation use-cases. In non-persistent binding mode, blocks must be explicitly requested by the GPU every time page replacement runs. In the case of PIC-like simulation, where page replacement runs before each frame, this makes the grid a scratchpad data-structure.

Grid data is undefined at the start of each frame, and persists only until the next frame begins. After requests are fulfilled, the request bitmap is fully reset to zero.

Naturally, persistent mode is not so transient. The block request bitmap is left unchanged after each run of the memory manager. Memory pages bound to blocks remain bound for the lifetime of the VSPgrid or until explicitly freed by GPU code flipping the corresponding bit. This mode does not typically have a use-case within PIC simulation, but would be valuable for other GPU volumetric applications.

VSPgrid's block request resolution and memory management routine can be initiated at any time through a host side VSPgrid member function. Both persistent and non-persistent mode use the same underlying algorithm to resolve memory needs. This process begins by computing two complimentary bitwise and operations between the current state bitmap and request bitmap. $\sim$`current & requested` computes a bitmap which shows all blocks which need a newly bound memory page, and the result of `current &` $\sim$`requested` similarly identifies all blocks whose pages can be freed.

This pair of resultant bitmaps is then used to identify the number of memory pages which should be 'moved', 'deallocated', or 'newly allocated'. Moved memory pages are pages bound to blocks which were previously needed, but are now marked for freeing. Instead of deallocating the page, these pages are rebound to unbacked blocks that have been requested. If the total number of requested blocks nets lower than the current count, the memory pages in the difference are deallocated. Conversely, if the net change in memory backed blocks is positive the memory pages in the difference are newly allocated in bulk.

We use the Vulkan Memory Allocator Library [6] to provide an allocator implementation atop of Vulkan API's bulk allocations. Each VSPgrid is granted its own dedicated pool of sparse binding compatible device local memory to allocate from

and free to. Since all memory pages are of equal size, and almost certainly a power of two, we choose to use the buddy allocation algorithm.

The final detail of note about VSPgrid's memory management scheme is its integration with Vulkan synchronization. The VSPgrid memory management function ultimately produces a batch of sparse binding instructions which must be executed by a compatible `VkQueue`. To avoid memory hazards and minimize the overhead of host-side memory management, submission of the binding instructions must be synchronized with the rest of the application.

VSPgrid tries to provide the caller of the memory management function with as much control over synchronization as possible. The memory management function described thus far does not actually submit any sparse binding instructions to the GPU. Instead it accumulates all sparse binding instructions into an opaque object representing 'pending bindings'. The caller can thus defer submission of the binding instructions as needed. When ready to submit, the caller populates a `VkBindSparseInfo` struct with the appropriate synchronization primitives. This structure, the sparse binding queue, and the 'pending bindings' object are then passed to a VSPgrid 'submit' function to finalize the submission. This gives the caller full control of when binding occurs, what dependent GPU work it triggers, and which queue processes the request.

5.5 GPU Compute Grid Management

Requiring GPU compute code to split itself into two steps so that it can request grid memory before accessing data may seem overburdening at first. However, the GLSL interface provided makes requesting access to grid data no more complicated than the access itself. By virtue of the sparse paged grid's design, accessing grid data from at a given coordinate can be accomplished by passing the coordinate to

the VSPgrid addressing function, then accessing the grid data array at the address returned. Requesting that a given grid coordinate be made active by the memory manager only requires passing that same address to a utility function. As a result, requesting access to grid data requires very little additional code.

To split GPU compute code into two steps, GPU code authors can generally choose between two options. For maximum simplicity in situations where the only GPU compute work to be done requires the sparse grid, authors can plan to run the same GPU compute code twice per frame. The author then wraps the first grid data access to any address with a check for the validity of that address. If the address is invalid, they request the address be activated and terminate the compute routine. The second run of the code will occur after grid memory is managed, and execute with the full compute instructions.

```
void gpuCompute(){
    ...
    GridBitpack cellAddr = pack_cell_coord(xyzCoord);
    if(is_cell_resident(cellAddr)){
        gridData[cellAddr] = newValue;
    }else{
        mark_for_activation(cellAddr);
        return;
    }
    ...
}
```

Figure 5.6: Grid compute code designed to be run twice.

The more preferable option, especially in cases where grid computation is not the only GPU accelerated computation being done, is to split grid logic into separate compute dispatches. The first dispatch requests activation of the needed blocks as early as possible then exits, leaving all grid computation to the second dispatch. This approach is advantageous as it harnesses concurrency between the host and GPU. By inserting non-grid GPU compute dispatches in-between the request and access steps, the GPU can remain productive while the host resolves grid requests. Furthermore, on hardware supporting asynchronous sparse binding and compute queues the entire

process of handling grid memory might be handled concurrently with other GPU tasks. This is the approach used by our example PIC-like simulation.

Chapter 6

PARTICLE ARRAY RESOURCES

In their simplest form, Lagrangian particles can be implemented easily with a contiguous array of structures. However, unlike the grid which is used as a scratchpad, Lagrangian particles in PIC-like simulations have additional responsibilities which complicate implementation. In PIC-like simulations, Lagrangian particles are the canonical representative for the state of all material within the simulation. Likewise, it is particle data which must be exported as the final product of the simulation. Throughout its lifetime, every frame of particle data is an output target of the simulation, an input for its successor, and the source for output to disk. This triplet of dependencies can greatly complicate the design of a high-performance PIC-like simulator, and have massive performance implications if mishandled.

This thesis' express goal is to drive a high-performance simulation framework using the computational power of the GPU. To achieve performance, any implementation must take care when handling particle data, however using the GPU to drive computation introduces further complications. For GPU PIC simulation, particle data must be read and written on the GPU, but only the CPU can handle writing it to disk. This introduces particle data dependency relationships between a host and GPU which operate asynchronously. It also introduces the need for an additional GPU to CPU transfer of particle data atop the existing write out to disk on every frame.

Logical complexity aside, the need for synchronization of particle data between CPU, GPU, and disk poses a fundamental threat to the efficacy of a high performance GPU implementation. It is commonly recognized that GPU driven applications can suffer greatly when memory transfers with the host are conducted in excess. Band-

width between the devices in limited, and may require synchronization that halts execution of GPU workloads. Given the GPU's massive parallel computational power, the speed with which results are computed within device local memory are likely to outstrip the rate at which those results can be handed off to the host.

As slow as host/device transfers are relative to GPU computation, the speed of disk I/O is all but guaranteed to be slower. As a result of both considerations, any implementation which tightly couples particle data generation with particle data output will be heavily IO-bound and under-utilize GPU compute resources. To mitigate this limitation, we designed both our particle data resources and main simulation loop to be extensively asynchronous. In doing so we partially decouple GPU simulation computations from particle data transfer and output, while also increasing the effective host/device transfer and disk write speeds. This design results in an overall throughput which is several times greater than the serialized alternative.

Due to the centrality of particle data to the structure of our framework's primary loop, the design of our particle data methods is an inextricable subset of the larger asynchronous simulation loop design. In this chapter we will detail this subset, and introduce elements from the broader design only when needed for context. In Chapter 7 we will unite all prior implementation chapters to describe the full design.

6.1 Implementation Overview

Internally, our particle data always takes the form of a long array of structures. Much like we wanted GPU code to view the sparse grid as if it were a dense array of grid cells, we also want GPU code to view particle data as an contiguous array of particle instances. In our implementation, GLSL compute code is entirely decoupled from the host-side complexities of particle data management. The only need for coordination between GPU particle compute code and host particle management is for the particle

data-type to match between languages. This is necessary only so that the host can meaningfully set the initial state of the simulation and encode particle data into its final output file format.

Our implementation exports final particle data results to disk as individual frames saved in Houdini™'s binary geometry format (.bgeo). Translation of our internal particle data to bgeo, and the subsequent output to disk, is accomplished using Walt-Disney Animation's open-source particle IO library: *Partio*[3]. Partio's internal representation differs from our own as they specify particle variables with separable attributes rather than a single structure. As a result, encoding of our particle data prior to disk writing requires translation into the Partio internal format, as well as a subsequent conversion to bgeo format. Future work may investigate taking a more direct route to disk output.

The most significant restriction of our particle data management is that it requires that the number of particles in a simulation be kept constant throughout the course of the simulation. More accurately, the quantity of memory allocated for the particle array must be constant for the lifetime of the simulation. It is not uncommon for PIC-like simulation models to maintain this assumption irrespective of implementation restrictions, as particles represent immutable quantities of matter. There are however material models which rely upon this capability, as they may merge or split particles during simulation. Additionally, some simulations may wish to emulate inflow or outflow of material from the simulation domain by adding and removing particles.

Although we would like to provide a more robust handling of variable particle counts in the future, our current implementation should still be capable of handling the above use cases through alternative methods. Since memory constancy is the only real requirement, PIC-like simulations can flag particles with any number of 'active' or similar state indicators. Particles which should be removed can be marked

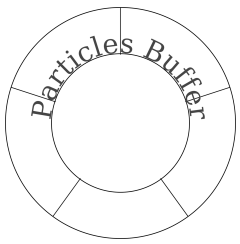
as inactive, and excluded from future computations. Likewise a portion of memory could be initialized with disabled particles and later made active to emulate the creation of new particles. This type of flagging strategy is not just a workaround for a lack of dynamic memory, but a valid solution it its own right, which likely has better performance than an implementation reliant on dynamic allocation of GPU memory.

The complexity of our particle data implementation comes from our enabling of asynchronous particle data usage. This asynchrony is accomplished primarily through extensive use of multi-buffering. Particle data used both on the GPU, as the input and output of a simulation frame, and on the CPU, as sources for copying to disk, are stored in ring buffers. These ring buffers maintain an auto-recycling backlog of particle data for previously simulated frames. The availability of this backlog allows for particle data transfer to the CPU and disk to occur concurrently without immediately blocking the continuing progress of the simulation.

6.1.1 Multi-Buffered Particle Data Management

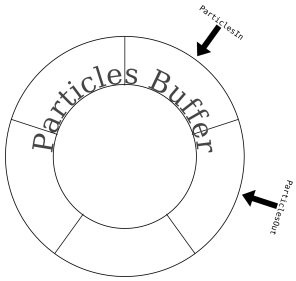
On the host's side, nearly all particle data is owned and managed by an instance of our `MultiBufferedParticleManager` class. Construction of this class requires a particle data-type and the total number of particles required for the simulation. All resource allocation is conducted at the time of construction, and no reallocation occurs during the instance's lifetime.

Rather than allocating a buffer with just enough space to contain the requested particle count, the class allocates a larger buffer of size $N * n_p$ where n_p is the number of particles and N is the amount of multi-buffering (whole number $N > 1$). The resulting allocation is a ring buffer of N many segments where each segment



$N = 5$

is an array of n_p particles. This ring buffer is allocated from device local memory heaps for optimal access from GPU compute code.



Integrating this ring buffer with GPU compute code is simple, and requires no awareness on behalf of the GPU code. Compute modules using the our GLSL particle interface are provided with two particle array bindings. The first is the `ParticlesIn` array, a read-only particle array containing the state of all particles as of the prior frame. The second is the `ParticlesOut` array, which is identical in size to `ParticlesIn`, but which is the writable target for the particle data processed by the current frame's GPU compute code.

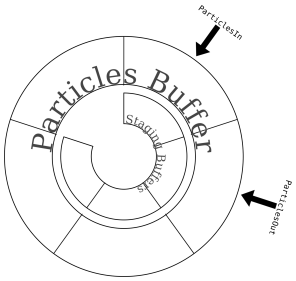
```
// particles.glslh
layout(..., std430) readonly buffer ReadParticleBuffer{
    Particle bParticlesIn[];
};

layout(..., std430) buffer WriteParticleBuffer{
    Particle bParticlesOut[];
};
```

Figure 6.1: GLSL particle array buffer bindings

When using Vulkan resource descriptors to bind Vulkan buffers to GPU compute pipelines, one may bind a sub-range of a larger buffer rather than bind the entire buffer range. Because our particle ring buffer is a contiguous device local storage buffer, we can redirect the `ParticlesIn` and `ParticlesOut` arrays to sub-ranges of the ring buffer. We use update-able descriptor bindings enabled by the optional Vulkan 1.2 feature: `descriptorBindingStorageBufferUpdateAfterBind`, to shift both particle array bindings by one ring buffer segment per-frame. The resulting 'rotation' around the ring buffer is such that the `ParticlesOut` array of the prior frame becomes the `ParticlesIn` array of the next. While simulation advances, $N-1$

frames of prior particle data remain concurrently available within the ring buffer and are automatically recycled as the rotation continues.



The device local particle ring buffer is only the first tier of multi-buffering in our implementation. Because this buffer is allocated from device local memory, it cannot be read by the host for the purposes of writing particle data to disk. To make the data available, the completed frames of particle data must first be copied into host coherent GPU memory. Thus we create a second tier of multi-buffering by creating a secondary 'Staging Ring' from a host coherent buffer. The elements of this second ring are the same size as the elements of the device local ring (n_p), but there is one fewer segment in the ring ($(N-1) * n_p$).

All of the elements of the staging ring are utilized as destinations for buffer copy transfer operations which run on the GPU. Once particle data has been downloaded into the staging ring through these operations, it can be directly accessed by the host via mapped memory. On platforms with a GPU supporting asynchronous memory operations, the download of particle data into the staging ring can be run asynchronously with the simulation of future frames.

The device local particle ring and host coherent staging ring represent the entirety of Vulkan resources owned and managed by the `MultiBufferedParticleManager` class. Outside of these resources, the class provides a plethora of utility functions abstracting the ring like nature and rotations of the buffers. In isolation the class accomplishes very little other than memory allocation and tracking of offsets for binding to ring elements. The class and its buffers must be integrated within our larger asynchronous simulation loop, and governed by external synchronization rules to achieve its full purpose.

6.1.2 External Particle Concurrency and Synchronization

Within our framework's simulation loop, the particle staging ring's $N-1$ segments are used for spooling completed frames of particle data. Similarly the N frames of backlogging within the device local ring spool data for the staging ring. As soon as frames of particle data have completed computation in the outer ring they can be consumed for download into the staging ring. Once again, on supporting hardware the production and consumption of frames my be handled asynchronously by the GPU.

Thread Pool

Frames downloaded into the staging ring are ready to be passed to Partio[3] for output to disk. However it is expected that this process will likely be several times slower than the time taken to produce each frame of particle data. Furthermore we do not wish to block the simulators main thread, as it is responsible for VSPgrid memory management and submission of all GPU workloads. As a remedy, we delegate the particle frame output task to a pool of worker threads. Using this thread pool prevents blocking IO, but also leverages CPU concurrency to parallelize the cost of encoding particle data prior to disk writing.

All of these components, device local ring, staging ring, and particle IO thread pool, cannot be expected to work in harmony without external coordination. The risk of race conditions exist at all levels of the presented system. GPU computation of future frames could overwrite data before it has been downloaded, and similarly the download of a newly completed frames might overwrite data which is in the process of being output to disk. Although all components are designed to operate asynchronously, their combined effort must be semi-synchronous. Their dependence on one another is dictated by the time taken to complete operations within all com-

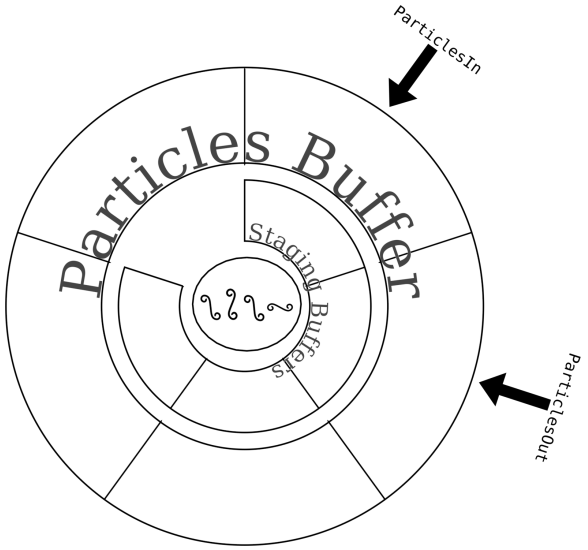


Figure 6.2: Particle ring buffers and thread pool

ponents and the degree to which multi-buffering allows them to continue operating before facing a data race.

Data race prevention is accomplished primarily through the use of two synchronization primitives: Vulkan fences (`VkFence`) and C++ mutex locks, which are used for protecting GPU and host resources respectively. Fences and mutex locks are associated with segments of the device local and staging ring buffers respectively, as is shown in Figure 6.3.

Although Vulkan fences are not mutual exclusion locks by design, we use them to achieve an analogous result and will discuss them as such. Segments of the outer ring are 'locked' anytime they are the target for GPU compute output, or are the source in a particle data download. Both operations must acquire the lock prior to executing, preventing any clashes. Note that although conceptually associated with segments of the device local ring, the fences described are created and owned by the simulation loop not the particle manager.

A similar segment-wise locking mechanism is used to protect the staging ring, but with an added notion of movable ownership. Each segment of the staging ring

is associated with a lock derived from the C++ `std::unique_lock<std::mutex>` template. This lock ensures it has only a single concurrent owner. By default, each mutex is owned by the particle manager and kept unlocked. Initiation of particle data download into a staging segment locks the segment, and passes ownership to the thread handling the frame's output. When the thread finishes reading the particle data, the segment is unlocked and returned to the manager.

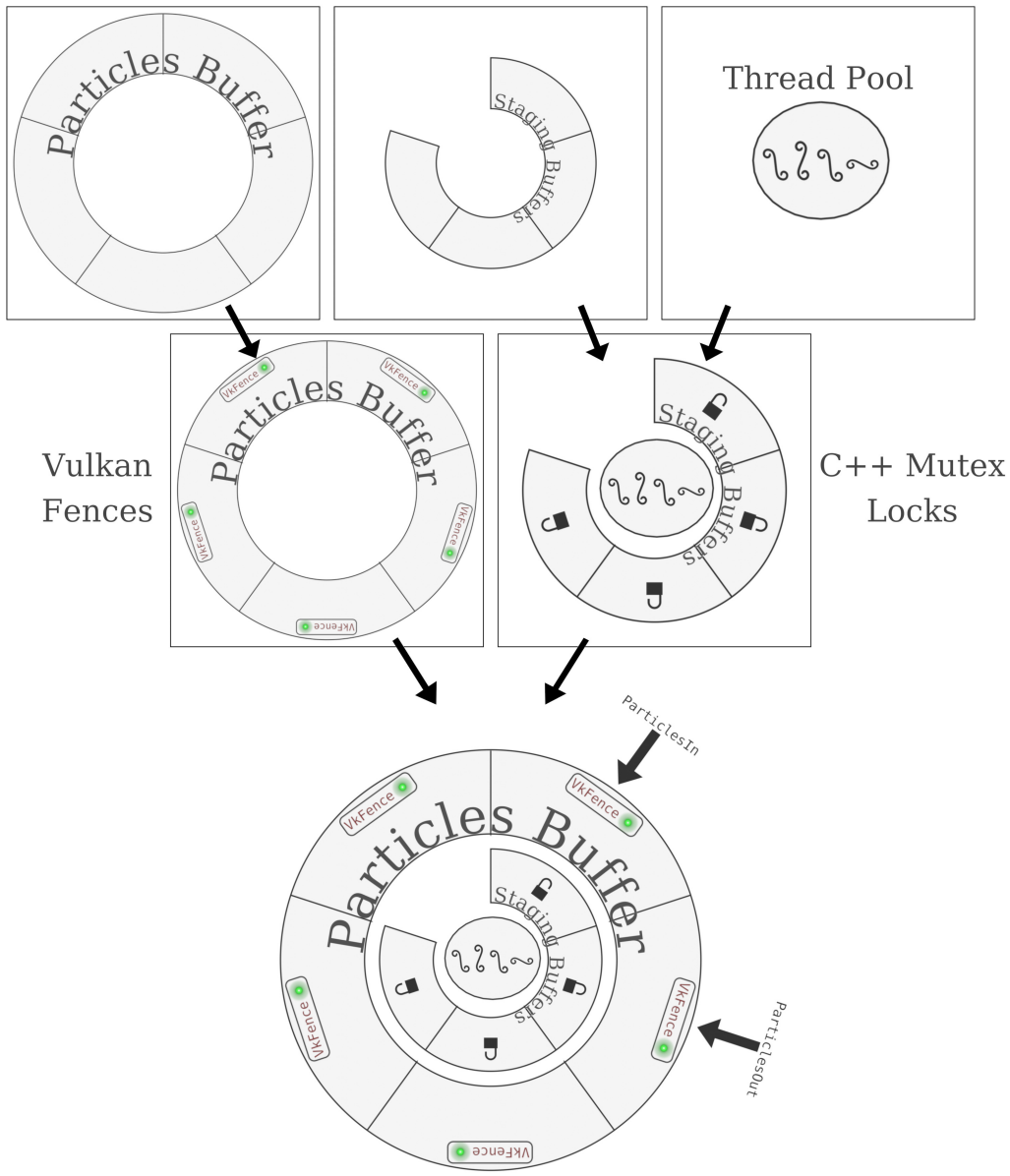


Figure 6.3: Complete assembly of asynchronous particle components

Figure 6.4 below provides a frame-by-frame illustration of how all of these components operate in coordination under idealized circumstances. As the simulation advances, the `ParticlesIn` and `ParticlesOut` references rotate about the outer ring and leave a trail of completed frames. These completed frames of particle data are concurrently propagated inward until they are finally written to disk.

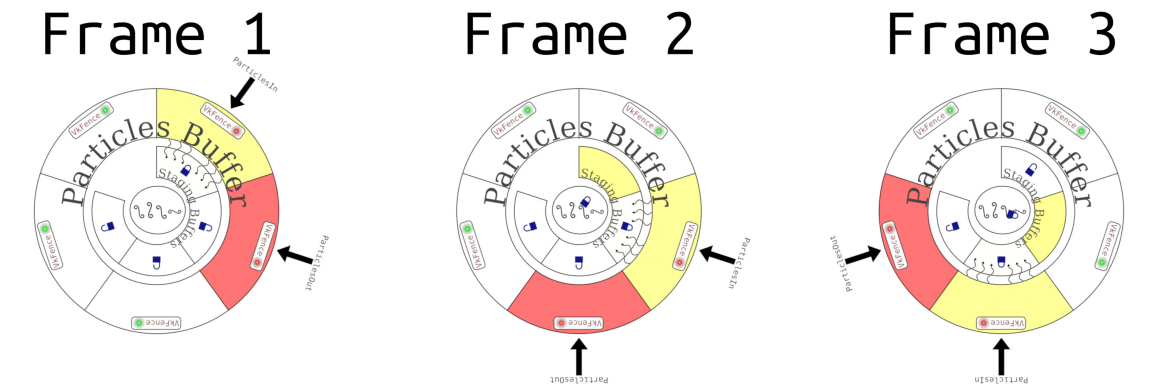


Figure 6.4: Illustration of the first three frames of asynchronous particle management operating with ideal timing.

At the start of a simulation, the first segment of the device local particles buffer is populated with the initial state of the particles in the simulation. This is marked by the yellow segment on the outer ring of frame 0. Yellow segments indicate that a segment is populated with particle data that the GPU has completed generating, but which has not yet been duplicated elsewhere. Red marks the segment which is being actively modified by simulation compute code.

Through the duration of frame 1, the yellow segment containing existing particle data is simultaneously accessed by GPU simulation code as the `particlesIn` array, and used as the source for a download into the first segment of the staging ring. Note that the `VkFence`s for both of the colored outer segments as well as the mutex lock for the first staging segment are all locked.

By the time the simulator moves onto frame 2, the first segment of the staging buffer is populated with the particle data transferred during the previous frame. Since

that data is now safely residing in the staging ring, its previous location in the outer ring is unlocked and may be overwritten. Ownership of the staging segment's lock has been transferred to the thread pool, and work to encode the frame is in progress. Simultaneously, the particle data resulting from the GPU compute work of frame 0 is being downloaded into the second segment of the staging ring.

By frame 3 the first particle data frame has been written to disk, and the lock for that stage has been released, and all other components continue uninterrupted in the same fashion as before.

The multi-buffered asynchronous nature of our particle management and IO results in far greater throughput. Specific results of this nature will be given in Chapter 8. Naturally, the benefits attained are dependent on the amount of multi-buffering (N) chosen. Higher values of N will typically improve the average simulation time, but will eventually approach a limit. When selecting an appropriate value a user of our framework must weigh the added memory footprint of a higher N value against the potential speedup. In our test cases we found that $N = 6$ tended to be the point after which diminishing returns made higher values unjustifiable.

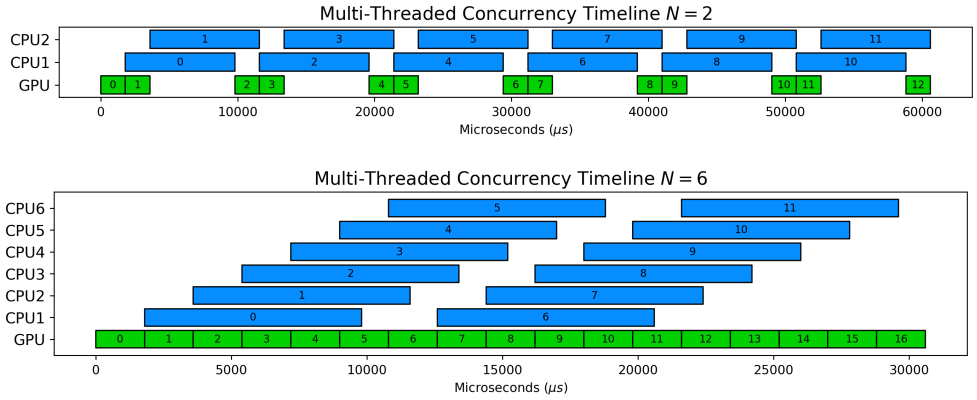


Figure 6.5: Conceptual comparison of concurrent timelines for $N = 2$ and $N = 6$ on ideal hardware configuration.

Chapter 7

PIC ASYNCHRONOUS SIMULATION LOOP

Due to their centrality and significant practical challenges, understanding the structure and maintenance of our grid and particle resources is necessary for understanding our framework. Now that those topics have been covered ad nauseam we can return to a more holistic view of the entire implementation. We will unite all of the concepts discussed so far to describe the primary loop of our implementation. A loop which, thanks to prior design choices, is able to focus on driving the continuous operation of many asynchronous components.

Recall that our simulation framework consumes PIC-like simulation specifications and executes them as independent simulations. Each PIC-like specification provides a list of GPU compute modules to be executed in sequence by the framework. Continued description of the framework and its simulation loop must begin with an explanation of how this sequence of compute modules are resolved into a collection of Vulkan GPU compute workloads. We must also discuss how the resulting PIC-like simulation pipeline gets tied into our grid and particle resources to ensure memory consistency, without creating excessive delays in the GPU's execution timeline.

7.1 Building a PIC Pipeline and Command Recording

Prior to running the main simulation loop, input PIC pipeline specifications must be used to configure simulation resources, produce command buffers, and generate instructions executable by the loop. This process is broken up into three basic steps: (1)PIC Pipeline Building, (2)Resource Configuration, and (3)Command Recording.

(1) PIC Pipeline Building handles the translation of the JSON pipeline specification into an internal representation. Each PIC pipeline is internally represented by a class possessing basic configuration variables, an optional reference to a particle initializer class, and a list of `PICstage` abstract class references.

The configuration variables consist of the number of frames to simulate, the frame-duration/framerate, simulation domain bounding box, and grid resolution. The optional particle initializer reference directs to an abstract class instance which may create and otherwise pre-condition particle data prior to simulation.

The `PICstage` list is the focal point of the internal PIC pipeline class. Each element of the list references the internal representation of a PIC GPU compute stage in the original JSON specification. The `PICstage` abstract base class allows for significant specialization of concrete classes if needed, but we presently handle all JSON specified stages with generic programmable stage classes. Our `ProgrammablePic[Type]Stage` classes, where '[Type]' is Particle, Grid, ect..., are designed to be compatible with any stage of the given type, and pull their GPU compute code from the SPIR-V module given by the associated JSON object. We also define a specialized `ReorderStage` class which represents a fixed function stage whose use is governed by the framework. Details of this stage are given in Section 7.2.

Each `PICstage`, regardless of the derived class, is "built" when the class is constructed. The key product of building a PIC stage is a Vulkan compute pipeline (`VkPipeline` and `VkPipelineLayout`). The use of the word "pipeline" is easy to confuse with our PIC-like simulation pipeline, but is completely unrelated in this case. The Vulkan compute pipeline object contains the final native GPU compute code generated from the provided SPIR-V module, and lays out the binding points used to attach GPU resources to the compute code.

When we build this compute pipeline object, we specialize the compute module using any specialization constants given by the JSON object. We determine the correct resource binding layout based purely on the type of stage being built. Once built, each compute pipeline object can be bound and dispatched using instructions within a Vulkan command buffer. The framework is designed such that each built stage will be built only once and re-used many times. Future research may indicate that re-building stages with altered specialization constants mid-simulation could provide performance benefits.

(2) Resource Configuration occurs second during setup of a PIC pipeline as it is dependent on the configuration values retrieved in the prior step. This step is responsible for initializing all simulation resources, including the particles and grid. Although the Vulkan API calls needed for this step are extensive, they are not unique to this work and thus will not be discussed.

The area of most interest is in the execution of any particle initializer specified by the PIC pipeline. As mentioned prior, the particle initializer is derived from an abstract initializer class, and has free reign to create and pre-condition particle data. For the initializer, the particle data is a vector of particle structures which it may freely add to, subtract from, or modify as necessary. At this time, the initializers provided support procedurally generating primitive spheres and rectangular solids. Most flexibly, a file load initializer allows arbitrary particle collections to be loaded from disk.

After initialization, the particle data is copied into a host coherent staging buffer, then copied to the device local particle buffer. There is no need for this type of initialization on the sparse grid as grid resources are allocated on demand. The only configuration details passed to the grid are the simulation domain bounding box, and requested grid resolution.

(3) Command Recording is the most involved step of PIC pipeline setup, and its design is inseparably linked to the architecture of the main simulation loop. This step begins by constructing an instance of a fixed function particle reordering stage. This is a necessary stage to enable all transfers between particle and grid data as will be expanded upon later in this section. Likewise, we automatically insert a reference to this reordering stage before any P2G stages, usually placing it at the start of each frame. After the reorder is inserted, the stage list is ready to be batched into workloads for the GPU.

The host application's goal now is to batch as much GPU compute work together as is possible. In this case "together" means as commands recorded into the same command buffer. The main obstacle is in managing demands on the sparse grid data-structure. All requests for sparse grid allocations originate from the GPU, but can only be resolved by the host application.

The solution is a loop through the stages which greedily consumes commands from each stage. Each run of the loop possesses an active `VkCommandBuffer` to record commands into, and a `VkSubmitInfo` object to configure the workload prior to submission. Each PICstage in the stage list is passed the active command buffer so that it may record its associated resource binding and compute dispatch commands.

Recall that recorded commands may execute out of order, thus we must insert barriers to ensure consistency. We insert barriers automatically between stages to protect from memory hazards. The type of each stage is typically sufficient in implying memory dependencies between stages, however we also require stages specify if their code will trigger a sparse grid update. When a stage indicates that it invalidates sparse grid state, we must record additional synchronization commands and terminate the recording on the command buffer.

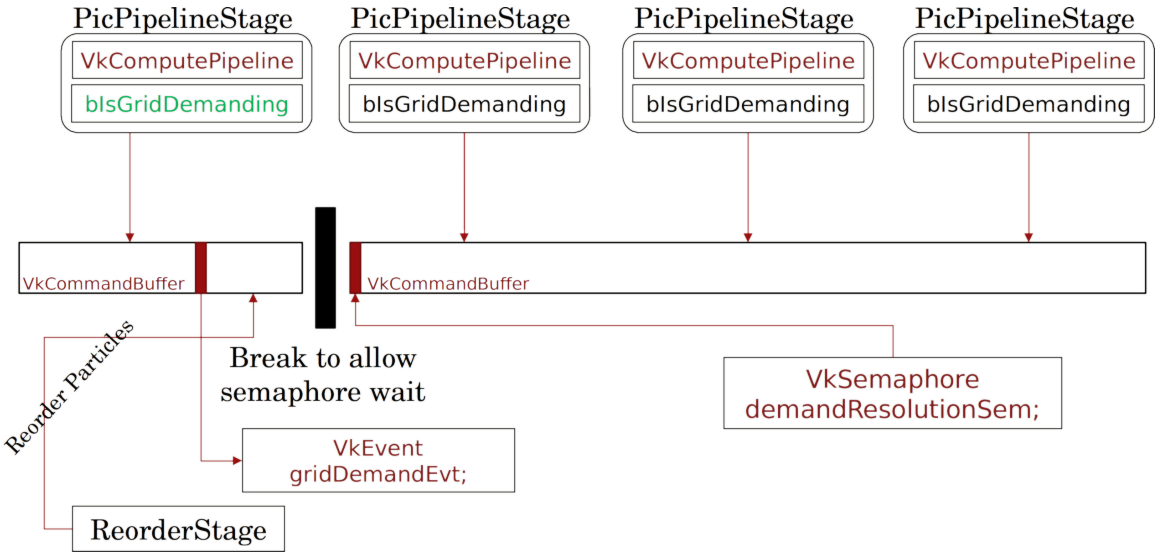


Figure 7.1: Command buffer recording and synchronization example.

After recording the commands for a stage which makes grid memory requests we append a `VkCmdSetEvent()` command into the active command buffer. Vulkan events are a synchronization primitive which are visible and modifiable from both the host (through API call) and device (through recorded command). Our framework maintains a single `VkEvent` whose purpose is to signal that requests for sparse grid resources have been written by the GPU. The host observes the state of the event object, and initiates grid management upon its occurrence.

Grid updates also typically imply the need for particle reordering and so the compute dispatches for that fixed function process may be appended to the command buffer as well. Once all supplementary commands have been recorded, we terminate recording of the active command buffer and begin the next. Splitting stages across command buffers in this way is designed to minimize delays between the completion of sparse grid memory management and subsequent simulation stages.

We record downstream pipeline stages into a fresh command buffer and make its execution dependent on the signalling of a dedicated "grid request resolution" semaphore. This semaphore is signalled by the sparse binding queue which fulfills

sparse memory binding workloads, and waited upon by the compute queue which dispatches simulation compute workloads.

Once workloads are submitted to a queue, the Vulkan implementation and device schedule execution independent of the host and resolve provided dependency relationships internally. As a result, using multiple command buffers and a semaphore allows for our host program to submit all compute stages at once without concern for synchronization. Grid dependent compute work will initiate automatically once sparse memory pages are bound.

7.2 Particle Reordering

There is a critical implementation detail for GPU PIC-like simulations which has been glossed over thus far. It is the transitioning of information between Eulerian grid and Lagrangian particles which, more than all else, characterizes PIC-like simulations. Yet we have not yet addressed how this is accomplished in a parallel context.

Recall that both P2G and G2P transfers in PIC-like simulations rely on the interpolation of information from many geometrically adjacent elements of one representation into adjacent elements of the other. As a consequence, all PIC-like simulations will require a spatial mapping linking particles, grid cells, and their respective neighbors. Furthermore, in a massively parallel context, grid elements updated during P2G tend to exhibit a very high data-race risk as most cells will be updated by many adjacent particles being processed by separate GPU threads. Effectively resolving the data-race also requires leveraging of a spatial mapping.

We learn our solution from Gao et al's work on GPU MPM[22]. As they do, we map between particles and grid cells by using spatial binning. This process leverages the existing spatially coherent organization of the VSPgrid to establish a particle to grid mapping without necessitating the construction of any additional data-structures.

Rather, particles are ordered within the particles arrays such that they follow the ordering of VSPgrid blocks. Each VSPgrid block containing an non-zero number of particles n, corresponds to a bin of n particles in the array. These bins are contiguous sub-ranges of the particle array, each of variable length.

Establishment of the spatial binning requires reordering of particles for every simulation frame. Generically, this reordering is a sorting problem in which particles are sorted relative to their location within the grid. The key value of the sort, is the index of the VSPgrid (sub-)block each particle resides within. The spatial coarseness of VSPgrid (sub-)blocks limits the number of unique values to be sorted relative to the total number of particles.

We choose GPU parallel prefix sort as the sorting algorithm with which to accomplish particle reordering. Parallel prefix sort applies well within the GPU context and has a mature history of GPU implementation guidelines. Additionally, the prefix sort leaves behind an important residual product, the prefix sum array.

The number of elements in the prefix sum array is the same as the number of (sub-)blocks within the VSPgrid. Likewise, there is a direct mapping between the (sub-)block indices and elements of the prefix sum array. When the parallel prefix scan step of the sort is completed, each element of the array stores the offset at which the corresponding (sub-)block's bin begins. Within the sorting algorithm, this array is used to determine the bin into which a particle is to be relocated. However after reordering is completed, the prefix sum array provides a direct mapping from blocks to their particles, and indicates the size of each bin through comparison with its neighbor.

Our implementation begins by filling what will eventually be the prefix sum array with the total number of particles in each bin. This fill is accomplished through a GPU parallel for loop iterating over all particles in the particle array. Within the

loop, bin sizes are incremented using atomic integer arithmetic. Once the individual sums are computed, a parallel prefix sum is conducted as a second phase.

Our implementation of the parallel prefix sort takes some algorithmic guidance from GPU Gems 3, Chapter 39[1], but reduces code complexity by leaning on the Vulkan® subgroup scan operation to coordinate threads. As in the source material, our implementation conducts several levels of localized prefix sum operations within threads. The results from the local sums are then combined to form the final prefix sum array. At the finest level, individual GPU threads scan over a small number of values, then consolidate their sums by shifting their totals to the right with a GLSL `subgroupExclusiveAdd()` operation. A similar non-subgroup method is used to consolidate all workgroup sums.

The final step of the reorder uses a GPU parallel for-each loop over all particles to move each to its new location within the particle array. The (sub-)block index for each particle is used to lookup an offset from the prefix sum array. Each lookup uses an atomic integer increment operation such that the retrieved offset is guaranteed unique. As a result of the atomic increments, the prefix sum array will have all of its values shifted one position to the left. However since the first element of the array is an implicit 0 offset, no valuable information is lost and the array remains a valuable utility for later stages.

With proper planning, the reordering stage can be integrated into existing stages and executed in parallel with other operations. In the typical PIC-like usage case, it is the position of particles within the simulation which dictates both sparse grid block activation and the need for reordering. Similarly, both an update of the sparse grid and particle reordering are pre-requisites of the P2G stage. The simulation stage which computes final particle positions within a simulation frame can, in the same routine, kickoff grid updates and reordering. Upon computing a particles new

position, its corresponding grid block index can be determined and used to both request updates to the sparse grid and compute the individual bin sizes for reordering.

Once the particle position update is complete, both the grid update and reordering processes can be started in parallel. Reordering executing as a compute job on the GPU, and a bulk of the sparse grid update work occurring on the host CPU. Furthermore, an implementation exposing asynchronous compute and memory transfer queues may be capable of continuing to execute reordering compute operations while also fulfilling sparse memory binding instructions.

7.3 Simulation Loop Anatomy

The complexity written into the implementation components discussed thus far allow for the simulation loop itself to be non-verbose despite the underlying logical complexity. The loop is written within a single 'run' function of the PIC simulator application class, and consists mostly of non-blocking calls which initiate work on the GPU and independent CPU worker threads. The loop exists primarily to arbitrate the scheduling of work and synchronize the disparate components of our framework.

The simulator's run function begins by creating a `VkFence` for each segment in the device local particle ring buffer. These fences are those discussed in Section 6.1.2. Despite it being conceptually useful to think of them as being owned by the particle manager, it is actually the loop code that must manage these fences in practice. The start of the run function also spawns the disk IO thread pool. The pool is formed using C++ standard library threads owned by a lightweight thread pool class. The pool consumes completed frames of particle data and delegates them to idle threads.

The simulation loop itself is a simple for loop over the range of 0 to N frames being simulated. Each run of the loop body simulates a single frame of particle data. The body of the loop begins by initiating the GPU $\rightarrow$ CPU download of the prior frames's

completed particle data. The 0th segment of the particle ring comes pre-loaded with the initial state of the particles, and so the first execution of the loop will use the 0th segment for `particlesIn`, and the 1st for `particlesOut`.

The fence and unique lock associated with the completed frame's segment of the particle ring are acquired first by the main thread. The download itself is then conducted on the GPU by an asynchronous transfer queue. The main thread never waits on this download to complete. Instead, it submits the unique lock and fence handle to the disk IO thread pool, where they are then delegated to the first free worker thread.

The worker thread waits on the provided fence prior to beginning export. The fence is signalled automatically once the asynchronous transfer queue completes its download. Since all waiting on the GPU occurs within the worker thread, the main thread is never blocked. By maintaining ownership of the unique lock, the worker thread prevents any modification of staging buffer memory being used to copy data to disk.

Once download and export of the prior frame has been initiated, the loop submits GPU compute workloads for simulating the next frame. An inner for-each loop iterates over the command buffers constructed from the built PIC-simulation pipeline. Before the inner loop executes, the main thread conceptually "takes ownership" of the `particlesOut` ring segment by waiting on and manually resetting the associated fence.

Within the body of the inner for-each loop, each command buffer in the pipeline is submitted to the asynchronous compute queue for execution. Since separate command buffers indicate that the VSPgrid must be updated, any non-last command buffer submission is followed a wait for the grid request event to be triggered. Once the event occurs, the appropriate VSPgrid functions are called to update the grid.

At this point in the loop, all work which can be done concurrently with VSPgrid management will have already been submitted to the GPU, or be operating in a worker thread. As a result, VSPgrid management is handled by the main host thread, and is the only significant simulation work conducted on the main thread. Recall that it is the host's responsibility to allocate GPU memory and determine how to bind that memory to a sparse buffer. However it is the GPU's responsibility to fulfill those binding instructions.

Once the host submits binding instructions to the GPU, the main host thread is able to continue, and will submit grid dependent workloads while sparse binding instructions are still being completed. No data-race is introduced as grid dependent GPU workloads will not be executed until the sparse binding queue completes and signals the grid request resolution semaphore.

At the tail end of the loop the `particlesIn` and `particlesOut` references are rotated one step around the ring by updating the appropriate resource descriptors. After exiting the loop, the run function submits the final frame of the simulation to the thread pool for export and waits for export to complete. Finally, all created synchronization primitives and the thread pool are destroyed.

Chapter 8

RESULTS & ANALYSIS

The contributions of this thesis, as stated in the introduction, are a Vulkan GPU sparse paged grid (VSPgrid), high-throughput multi-buffered particle arrays, a programmable GPU compute pipeline for PIC-like simulations, and an asynchronous simulation loop. The results of the first two will be described independently with respect to both their role in the framework, and for their efficacy as a standalone utility. The latter two contributions will be discussed jointly as critical elements in our final PIC simulation framework.

8.1 VSPgrid

Our implementation of a Vulkan GPU sparse paged grid is a direct adaptation of the original CPU sparse paged grid. We aimed to be as faithful to the virtues of the original as possible. We achieve a GPU local memory sparse paged grid utilizing Vulkan's sparse buffer feature set. This accomplishes the creation of a tree-less sparse grid with a completely dynamic memory footprint and dynamic topology by GPU demand. While the GPU cannot be totally excluded from the management of the sparse grid, its relationship is highly simplified and computationally negligible, there being no logical difference between accessing grid locations and requesting they be allocated.

Separate from the grid's function in an active PIC simulation pipeline, we validate VSPgrid through a suite of unit and coverage tests. Passing these tests demonstrates the VSPgrid as feature complete and stable with respect to different configurations and use cases. Tests are run fully independent of all other implementation compo-

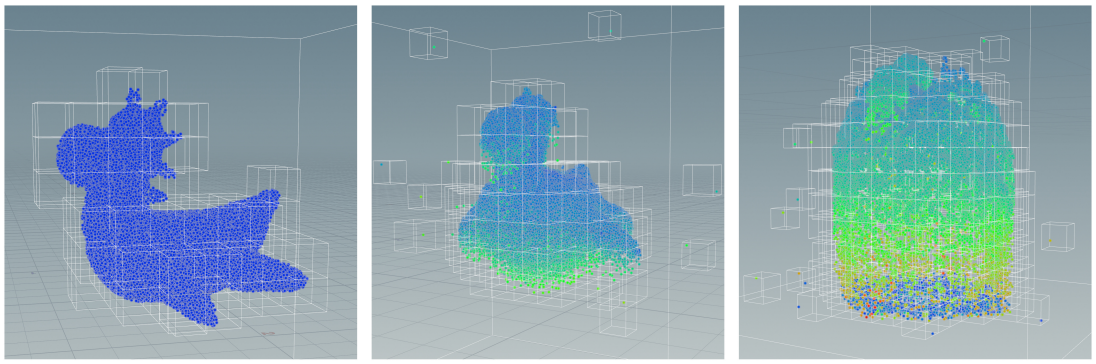


Figure 8.1: Grid visualization showing the evolution of VSPgrid active block topology as simulation advances.

nents, further demonstrating our VSPgrid's potential to function as a general purpose sparse grid data-structure within other Vulkan applications.

An initial set of tests validates the mirroring of grid configuration constants between the CPU and GPU code. For each grid configuration in the testing set, a Vulkan compute pipeline utilizing our VSPgrid header file is built. Execution of the contained compute code then mirrors all GPU grid constants, both passed as specializations constants and derived thereof, into a host coherent testing buffer. The contents of this buffer is then downloaded by the host unit-testing code and validated by direct comparison with the host-side grid configuration values.

We also validate the VSPgrid addressing scheme utilizing the same grid configuration constants as mirrored between CPU and GPU. A basic test encodes and decodes a handful of grid address locations on both CPU and GPU side, then compares them via a feedback buffer. We also conduct an addressing coverage test by using a compute shader to iterate over all grid cells in 3D index space, encode their individual addresses, and atomically increment a value in a write-back buffer. Each grid cell increments the value corresponding to the index of the sub-block containing the cell. We validate the write-back buffer by verifying that the proper number of sub-block values exist, and that they all equal precisely the number of contained cells.

More critical are tests verifying the grids ability to support reading, writing, and grid topology updates. Once again we start simple, beginning with tests that request activation of a set of grid locations, then confirm GPU visibility and access to the newly allocated blocks. Confirmation is similar to prior tests. After block allocation occurs, a second compute pass initiates GPU threads to write their unique values to the grid, then read-back grid values and mirror them into a host accessible buffer. The host reads the buffer to confirm that all threads reported in successfully.

We also implement a grid coverage and random access test. The coverage test runs sequentially through the full range of the grid using multiple GPU compute dispatches to eventually touch the full grid with GPU threads. Demand for grid block allocation is handled as it arises, and results from grid blocks are once again mirrored and validated.

The random access test validates the VSPgrid's support for complete memory dynamism. The test runs a CPU side loop which selects a random grid coordinate and value on each run. The coordinate and value are then passed as a parameter for a compute dispatch which will attempt to write the value to the specified coordinate. If the specified location exists within an inactive grid block, the block is activated and the dispatch is run again to complete the write. After each successful write into the grid, the value is read-back into a feedback buffer at the index of the current loop iteration. When the loop completes, the feedback buffer is validated against a CPU side record of all the random values written.

The random access test is run on a VSPgrid in persistent mode, ensuring activated blocks will not be automatically deactivated. This allows the test to validate the grid's ability to handle both page hits and page faults within the grid's virtual address space. The test loop runs a number of iterations equal to twice the largest dimension of the

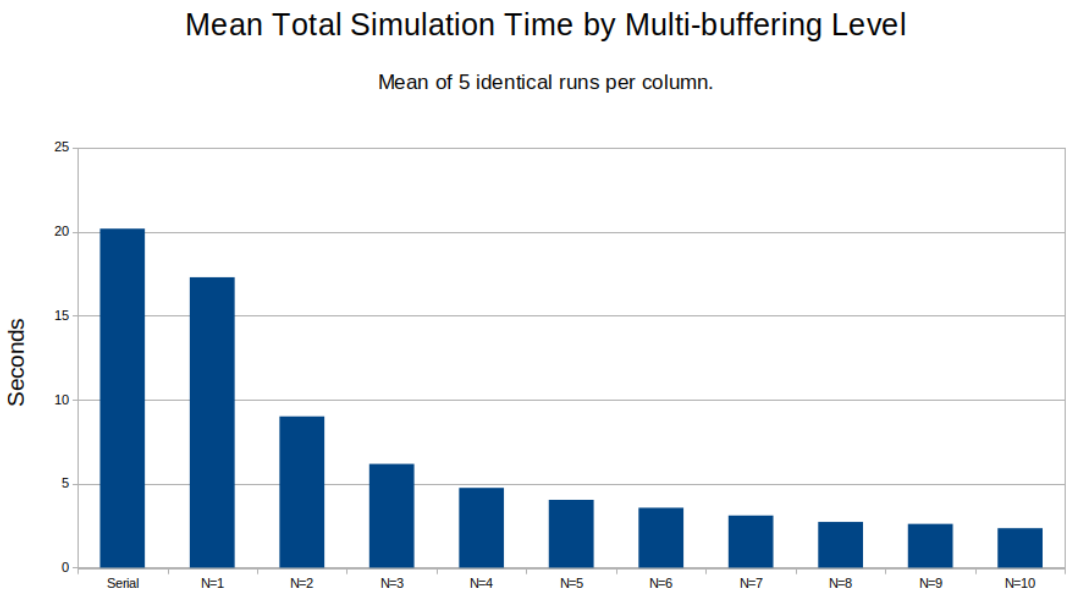


Figure 8.2: Average Total simulation time with increasing multi-buffering.

grid to allow significant random spread, while also mitigating the risk of running out of GPU memory by allocating a fully dense grid.

8.2 Multi-Buffered Particle Arrays

Our use of multi-buffered particle arrays mitigates the effect of a fundamental limitation of GPU driven non-realtime simulation. On high-end discrete GPU, the throughput of data accessed and produced by GPU computation will outpace the transfer rate of PCI-e and disk output in all but the most extreme cases. No matter how good a GPU PIC-like implementation may be, failure to acknowledge this limitation will result in sub-optimal performance.

Our ring buffered particle arrays present on both CPU and GPU minimize the transfer and disk bottleneck by increasing workload continuity on all fronts. The availability of buffered particle data from prior frames frees the GPU to continue its primary compute goals, while CPU cores which would otherwise be underutilized

work in parallel to clear the back-buffer. Until the ring buffers overflow, no operation will block the GPU or main CPU thread.

When comparing the overall simulation time of identical simulation configurations run with different levels of multi-buffering, we observe major increases in overall throughput. Improvements predictably follow a $\frac{k}{N}$ progression where k is a constant and N is the number of segments in the GPU local particle ring buffer. The return on adding additional segments inevitably diminishes, but not before yielding between a two and eight times improvement over serialized particle output. Profiling runs of our framework shows consistent saturation of workloads across worker threads in the particle output thread pool.

8.3 PIC-like Simulation Framework

We implement a baseline PIC-like simulation to validate our framework. We don't implement a physically based material model, instead implementing simple kinematics in which momentum is updated on the grid, and used to advect particle positions. The resulting simulation possesses all of the hallmark traits of PIC-like simulation, but lacks a solver and material specific mechanics. We use a quadratic B-spline weighting function for our particle/grid transfers.

We begin by presenting the results of a simulation which uses modest grid and particle resources. We simulate the exploding of the Houdini™ toy dinosaur mesh using our framework and test PIC implementation. The toy is made up of 57,332 particles, and simulated on a 256×256×256 Vulkan GPU sparse paged grid. Both resources are relatively low cost, the grid being small enough to allocate densely were a sparse grid not available. This simulation is shown visualized in the Houdini™ viewport by Figure 8.3.

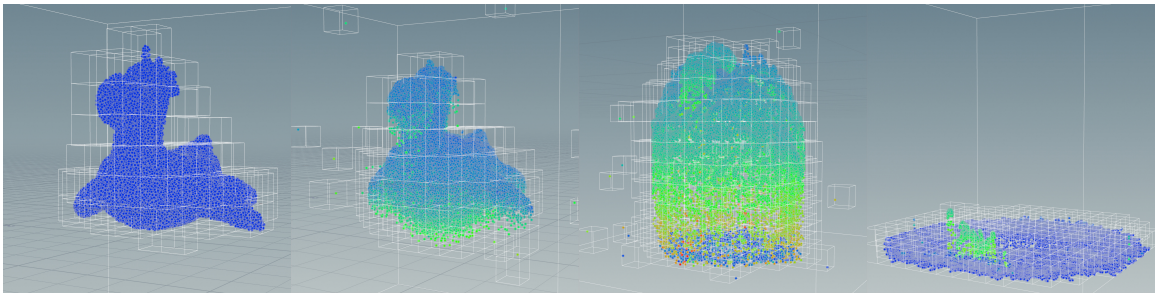


Figure 8.3: Exploding a toy dinosaur made from 57,332 particles on a 256^3 sized grid. Each box drawn maps to a VSPgrid block containing $16 \times 16 \times 16$ individual cells and mapped via virtual address space to a 16KiB memory page.

Given the relatively light workload of this initial simulation, our framework is able to complete simulation very rapidly. We render 512 frames for an animation which should play at 60 frames per-second. When using our default multi-buffering level of 5, the simulation completes in approximately 4.3 seconds, which is two times faster than real-time. If we increase the amount of multi-buffering we find that the peak speed for this simulation is about 2.2 seconds, which is four times faster than real-time.

When the simulation compute load is low enough to fill the particle array buffers, CPU worker threads are consistently saturated with work. This is shown by the CPU profiling timeline in Figure 8.4. Note that after the initial setup, activity on the main thread is extremely low. This is because the mid-simulation management of the GPU and VSPgrid is low impact work which frequently waits for GPU operations to complete. There is an excess of CPU resources in this case, so no benefit would be gained by moving additional work to the main thread.

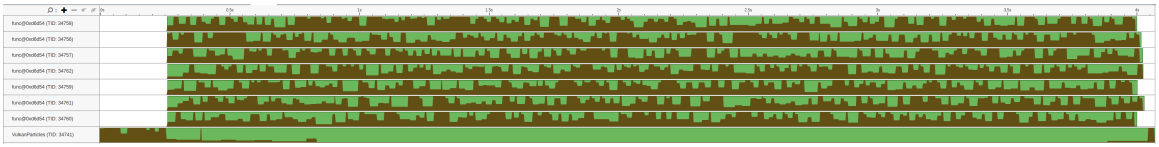


Figure 8.4: CPU thread activity timeline for the exploding toy dinosaur simulation, profiled with Intel VTune

Our second demonstration steps up simulation complexity to a more substantive level. We simulate a 3D Vulkan logo dissolving into particles as it falls to the floor. The simulation uses the same PIC pipeline as the previous, but with parameters adjusted via JSON specification and specialization constants.

The simulation operates on nearly a million particles, and runs with a 512×512×512 sparse grid. Were it to be densely allocated, this grid would consume 2Gb of GPU memory, a significant portion of total memory on many modern high-end GPU. Figure 8.5 shows frames from the simulated sequence. On average, we simulate the 256 frames in 27.43 seconds.

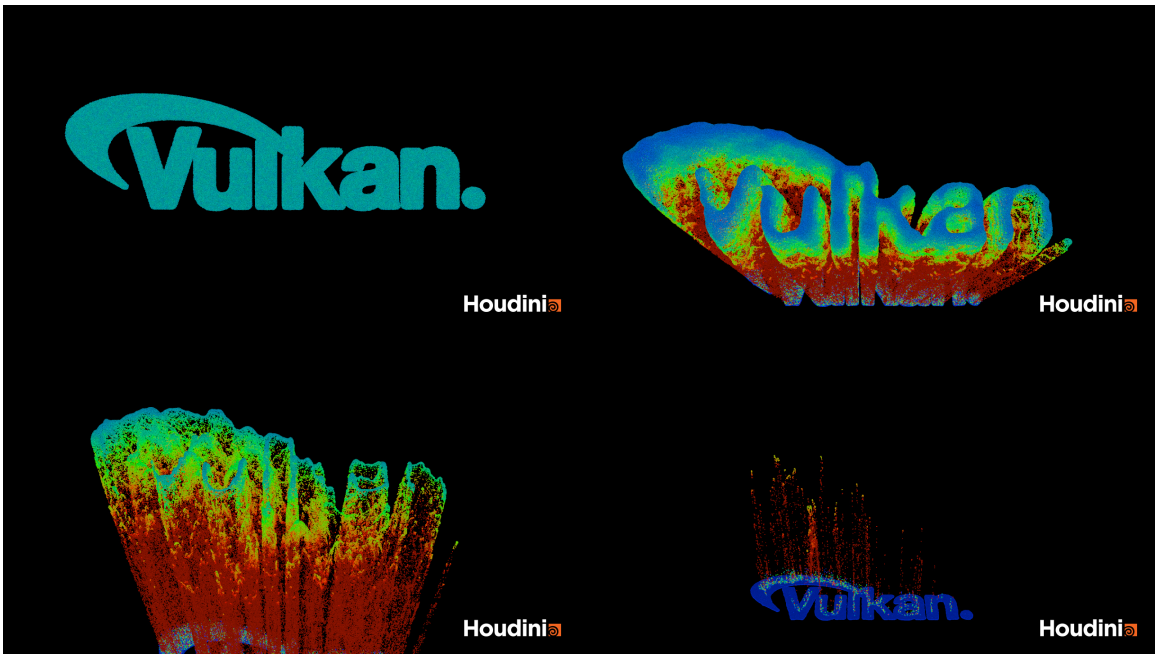


Figure 8.5: Dissolving Vulkan logo made from over 979,000 particles on a 512^3 sized grid. 256 frames rendered in 27.4 seconds.

We test VSPgrid's scalability and ability to extend beyond the range of physical GPU memory in a final simulation. This simulation is the same as the previous, but uses a 1024×1024×1024 grid. If densely allocated, this grid would require 16Gb of GPU memory. This is twice the available memory on our hardware, and exceeds or matches the GPU local memory on most high-end discrete GPU. The visual results

of this simulation are shown by Figure 8.6, and a comparison on the VSPgrid block granularity between the two simulations is shown by Figure 8.7.

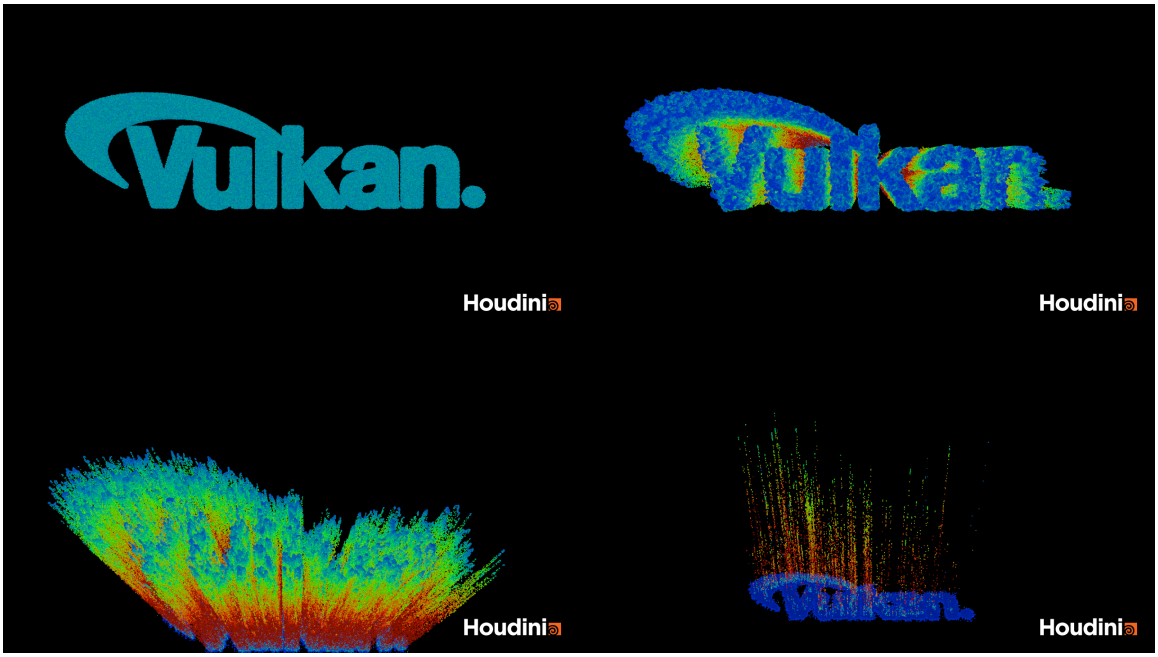


Figure 8.6: Dissolving Vulkan logo made from over 979,000 particles on a 1024^3 sized grid. 256 frames rendered in 26.50 seconds.

The total simulation time is practically unchanged due to the cost of increasing grid resolution being less than existing costs. GPU profiling shows that increasing the grid's resolution increased GPU compute time spent on particle sorting by between 3 and 5 times. This may sound like a lot, but has little impact due to the already low footprint of reordering. Reordering runs in the high-resolution logo drop simulation were recorded as taking between 18 and 25 milliseconds per frame, which is minor relative to particle/grid transfer stages which take about 205 milliseconds per frame. It appears also that with this number of particles the simulation will be bound by particle IO despite mitigation.

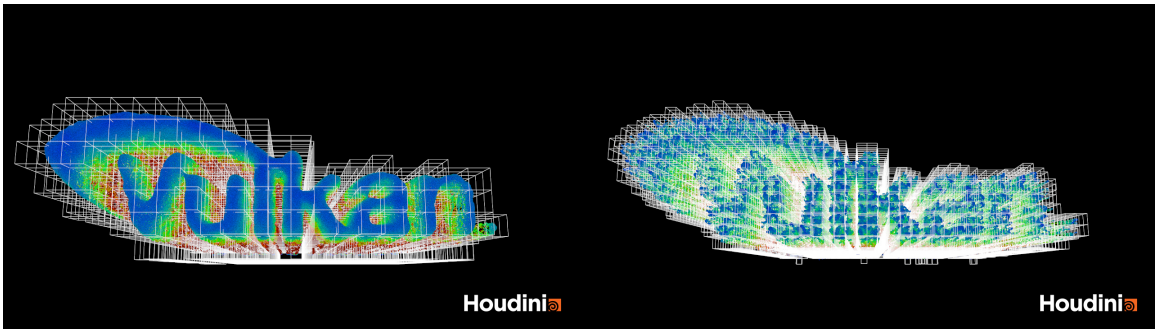


Figure 8.7: Left: Visualization of allocated VSPgrid blocks for the 512^3 simulation. Right: Visualization of allocated VSPgrid blocks for the 1024^3 simulation. Blocks in both images map to 16Kb pages and contain 4096 individual grid cells.

Chapter 9

CONCLUSION

We have presented a high-performance GPU compute framework for PIC-like simulation implemented using the Vulkan® API. Our implementation features a simulation loop which is broadly asynchronous, harnessing Vulkan's design to keep the GPU saturated with compute work. Integration of our asynchronous design with a data-oriented approach to PIC-like simulation particle and grid resources mitigates the bottle-necking effects which commonly plague GPGPU applications with a high-volume output data.

We also present a novel GPU adaptation of the Sparse Paged Grid data structure, restoring some of its original design virtues compared to previous GPU implementations. Our Vulkan Sparse Paged Grid has a fully dynamic memory footprint, and minimized GPU code involvement in grid management by reducing the relationship to an index keyed request then access scheme.

Multi-buffered particle arrays present on CPU and GPU and deeply integrated with our asynchronous simulation loop optimize particle data outflow. We implement this multi-buffering in acknowledgment of the critical factor PCI-e data transfer and disk IO speeds can play in a non-realtime GPU particle simulation. By utilizing async resources which might otherwise go unused, we prevent particle IO from disrupting simulation workloads whenever possible, and observe an up to eight times improvement in overall simulation speed relative to direct particle output.

Our framework exposes a programmable pipeline of GPU compute modules given automatic access to VSPgrid and multi-buffered particle resources. Pipeline configurations can be specified using human edited JSON describing an arbitrary sequence of

SPIR-V modules optionally customized with specialization constants. Each module being identified with a PIC-like simulation stage type, they are automatically integrated with the appropriate resources and synchronization of our implementation. This includes automatic insertion of a GPU parallel particle reordering step which provides a direct mapping between particle and grid perspectives.

9.1 Limitations and Future-Work

Our implementation is not without limitations, and there are still many opportunities for future enhancements and expansions. We believe that all of our contributions can and should be improved with future research and development.

Both our grid and particle array implementations currently use an 'Array of Structures' memory layout, which is broadly considered to be sub-optimal for memory performance in GPU compute applications such as ours. Fortunately, switching between layouts should be fairly non-intrusive and requires no major algorithmic changes. We would like to make future revisions of both resources utilize a data-channel based 'Structure of Arrays' layout which should provide general GPU performance benefits and be easier to configure a run-time.

We also hope to explore methods for increasing the dynamism of both grid and particle resources. The number of particles in a simulation is currently fixed for the entire lifetime of the simulation. Although it is typical for PIC-like simulation to maintain a fixed particle count, there are some models which allow for addition and removal of particles either as material inflow or particle merging and splitting. Although any per-frame dynamic allocation of particle memory would be unjustifiably inefficient, we would like to support limited addition and removal of particles. This is likely best achieved by pre-allocating additional particle memory and using bitmasking to selectively enable and disable particles.

In regards to VSPgrid dynamism, our grids size and resolution is freely configurable, but also fixed through the duration of the simulation. We would like to explore the possibility of adding adaptive resolution capabilities to the VSPgrid datastructure to allow selectively fine-grained computations, which is another feature of the original SPgrid. It may also be valuable to associate each VSPgrid instance with an arbitrary and mutable affine transformation allowing for more configurable embedding into worldspace. This would be similar to the method OpenVDB[43] uses to map VDB trees as world grids.

Regarding the rest of the framework, we believe that there is still room for improvement in several areas. There are still several parts of our simulation pipeline build process which uses overly coarse synchronization primitives, and may unnecessarily delay execution. There are also several areas in our provided GPU code which use inefficient methods for the sake of brevity and should be replaced. Our parallel prefix sort for example is more than sufficient, but not the state of the art for GPU parallel sorting. However, none of these changes require any fundamental restructuring of the framework.

A lack of Vulkan and SPIR-V libraries for mathematics and scientific computing are also an unfortunate limitation for our framework. Being both young and not directly tailored towards scientific computing, there are few opportunities to accelerate development by integrating third-party solutions to common problems. This will make implementing more complex PIC-like methods a more time intensive endeavour. To alleviate this pain as much as possible, we leave it as future work to expand our offering of utilities and template GPU code to provide well tested and tuned solutions to common PIC challenges.

BIBLIOGRAPHY

[1] Chapter 39. parallel prefix sum (scan) with cuda. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda.

[2] glslang. https://github.com/KhronosGroup/glslang.

[3] Partio - a library for particle io and manipulation. https://github.com/wdas/partio/.

[4] Shaderc. https://github.com/google/shaderc.

[5] Uintah. http://uintah.utah.edu/.

[6] Vulkan® memory allocator. https://gpuopen.com/vulkan-memory-allocator/.

[7] R. Abeyaratne. Continuum mechanics. *Volume II of Lecture notes on The Mechanics of Elastic Solids*, pages 1–100, 2012.

[8] S. E. Anderson. Bit twiddling hacks. https://graphics.stanford.edu/~seander/bithacks.html.

[9] S. G. Bardenhagen and E. M. Kober. The generalized interpolation material point method. *Computer Modeling in Engineering and Sciences*, 5(6):477–496, 2004.

[10] J. U. Brackbill, D. B. Kothe, and H. M. Ruppel. Flip: a low-dissipation, particle-in-cell method for fluid flow. *Computer Physics Communications*, 48(1):25–38, 1988.

[11] R. E. Bridson. *Computational aspects of dynamic surfaces*. PhD thesis, Stanford University, 2003.

[12] Q. Chen and H. Sui. Real-time particle-based snow simulation with vulkan. `github.com/giaosame/RealTimeParticleBasedSnowSimulation`, 2020.

[13] W.-F. Chiang, M. DeLisi, T. Hummel, T. Prete, K. Tew, M. Hall, P. Wallstedt, and J. Guilkey. Gpu acceleration of the generalized interpolation material point method. In *Symp App Accel High Perf Comp*, 2009.

[14] N. S. Cory Perry. Nvidia developer blog - introducing low-level gpu virtual memory management, April 2020.

[15] S. Cummins and J. Brackbill. An implicit particle-in-cell method for granular materials. *Journal of Computational Physics*, 180(2):506–548, 2002.

[16] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, 1996.

[17] J. Donea, A. Huerta, J.-P. Ponthot, and A. Rodríguez-Ferran. Arbitrary lagrangian–eulerian methods. *Encyclopedia of Computational Mechanics Second Edition*, pages 1–23, 2017.

[18] Y. Dong, D. Wang, and M. F. Randolph. A gpu parallel computing strategy for the material point method. *Computers and Geotechnics*, 66:31–38, 2015.

[19] G. Filipič. Principles of particle in cell simulations. 2008. `http://mafija.fmf.uni-lj.si/seminar/files/2007_2008/Seminar2.pdf`.

[20] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. Adaptively sampled distance fields: A general representation of shape for computer

graphics. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 249–254, 2000.

[21] M. Gao. *Sparse Paged Grid and its Applications to Adaptivity and Material Point Method in Physics Based Simulations*. The University of Wisconsin-Madison, 2018.

[22] M. Gao, X. Wang, K. Wu, A. Pradhana, E. Sifakis, C. Yuksel, and C. Jiang. Gpu optimization of material point methods. *ACM Transactions on Graphics (TOG)*, 37(6):1–12, 2018.

[23] J. D. d. S. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson. Uintah: A massively parallel problem solving environment. In *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*, pages 33–41. IEEE, 2000.

[24] P. Goswami, C. Markowicz, and A. Hassan. Real-time particle-based snow simulation on the gpu. In *EGPGV 2019*. Eurographics-European Association for Computer Graphics, 2019.

[25] S. I. Gunadi and P. Yugopuspito. Real-time gpu-based sph fluid simulation using vulkan and opengl compute shaders. In *2018 4th International Conference on Science and Technology (ICST)*, pages 1–6. IEEE, 2018.

[26] F. H. Harlow. The particle-in-cell method for numerical solution of problems in fluid dynamics. Technical report, Los Alamos Scientific Lab., N. Mex., 1962.

[27] S. Hodes. Leveraging asynchronous queues for concurrent execution, December 2016. `https://gpuopen.com/learn/concurrent-execution-asynchronous-queues/`.

[28] S. Hodes and A. Dunn. Deep dive: Asynchronous compute. Game Developers Conference, 2017. https://gpuopen.com/wp-content/uploads/2017/03/GDC2017-Asynchronous-Compute-Deep-Dive.pdf.

[29] B. Houston, M. B. Nielsen, C. Batty, O. Nilsson, and K. Museth. Hierarchical rle level set: A compact and versatile deformable surface representation. *ACM Transactions on Graphics (TOG)*, 25(1):151–175, 2006.

[30] B. Houston, M. Wiebe, and C. Batty. Rle sparse level sets. In *ACM SIGGRAPH 2004 Sketches*, page 137. 2004.

[31] Y. Hu, Y. Fang, Z. Ge, Z. Qu, Y. Zhu, A. Pradhana, and C. Jiang. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Transactions on Graphics (TOG)*, 37(4):1–14, 2018.

[32] Y. Hu, X. Zhang, M. Gao, and C. Jiang. On hybrid lagrangian-eulerian simulation methods: practical notes and high-performance aspects. In *ACM SIGGRAPH 2019 Courses*, pages 1–246. 2019.

[33] C. Jiang, C. Schroeder, A. Selle, J. Teran, and A. Stomakhin. The affine particle-in-cell method. *ACM Transactions on Graphics (TOG)*, 34(4):1–10, 2015.

[34] C. Jiang, C. Schroeder, J. Teran, A. Stomakhin, and A. Selle. The material point method for simulating continuum materials. In *ACM SIGGRAPH 2016 Courses*, pages 1–52. 2016.

[35] R. R. John Kessenich, Dave Baldwin. The opengl® shading language version 4.60.7, July 2019. https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

[36] Khronos Group. *More on Vulkan and SPIR-V: The future of high-performance graphics*. GDC, 2015.

[37] G. Klár, T. Gast, A. Pradhana, C. Fu, C. Schroeder, C. Jiang, and J. Teran. Drucker-prager elastoplasticity for sand animation. *ACM Transactions on Graphics (TOG)*, 35(4):1–12, 2016.

[38] F. Losasso, F. Gibou, and R. Fedkiw. Simulating water and smoke with an octree data structure. In *ACM SIGGRAPH 2004 Papers*, pages 457–462. 2004.

[39] J. Ma, H. Lu, B. Wang, S. Roy, R. Hornung, A. Wissink, and R. Komanduri. Multiscale simulations using generalized interpolation material point (gimp) method and samrai parallel processing. *Computer Modeling in Engineering & Sciences*, 8(2):135–152, 2005.

[40] M. Maldacker. Vortex2d. `https://github.com/mmaldacker/Vortex2D`, 2016.

[41] S. Markidis and G. Lapenta. The energy conserving particle-in-cell method. *Journal of Computational Physics*, 230(18):7037–7052, 2011.

[42] K. Museth. Vdb: High-resolution sparse volumes with dynamic topology. *ACM transactions on graphics (TOG)*, 32(3):1–22, 2013.

[43] K. Museth, P. Cucka, M. Aldén, and D. Hill. Openvdb. `https://www.openvdb.org/`.

[44] M. B. Nielsen and K. Museth. Dynamic tubular grid: An efficient data structure and algorithms for high resolution level sets. *Journal of Scientific Computing*, 26(3):261–299, 2006.

[45] D. Peng, B. Merriman, S. Osher, H. Zhao, and M. Kang. A pde-based fast local level set method. *Journal of computational physics*, 155(2):410–438, 1999.

[46] R. Setaluri, M. Aanjaneya, S. Bauer, and E. Sifakis. Spgrid: A sparse paged grid structure applied to adaptive smoke simulation. *ACM Transactions on Graphics (TOG)*, 33(6):1–12, 2014.

[47] A. Stomakhin, C. Schroeder, L. Chai, J. Teran, and A. Selle. A material point method for snow simulation. *ACM Transactions on Graphics (TOG)*, 32(4):1–10, 2013.

[48] J. Strain. Fast tree-based redistancing for level set computations. *Journal of Computational Physics*, 152(2):664–686, 1999.

[49] J. Strain. Tree methods for moving interfaces. *Journal of Computational Physics*, 151(2):616–648, 1999.

[50] D. Sulsky, Z. Chen, and H. L. Schreyer. A particle method for history-dependent materials. *Computer methods in applied mechanics and engineering*, 118(1-2):179–196, 1994.

[51] D. Sulsky, S.-J. Zhou, and H. L. Schreyer. Application of a particle-in-cell method to solid mechanics. *Computer physics communications*, 87(1-2):236–252, 1995.

[52] D. Tskhakaya, K. Matyash, R. Schneider, and F. Taccogna. The particle-in-cell method. *Contributions to Plasma Physics*, 47(8-9):563–594, 2007.

[53] S. Willems. Unofficial vulkan hardware database. `https://vulkan.gpuinfo.org/`.

[54] J. Wolper, Y. Fang, M. Li, J. Lu, M. Gao, and C. Jiang. Cd-mpm: continuum damage material point methods for dynamic fracture animation. *ACM Transactions on Graphics (TOG)*, 38(4):1–15, 2019.

[55] Y. Yue, B. Smith, C. Batty, C. Zheng, and E. Grinspun. Continuum foam: A material point method for shear-dependent flows. *ACM Transactions on Graphics (TOG)*, 34(5):1–20, 2015.

[56] Y. Zhang, B. Solenthaler, and R. Pajarola. Adaptive sampling and rendering of fluids on the gpu. 2008.

APPENDICES

Appendix A

MORTON ENCODING GLSL IMPLEMENTATION

These functions are written in Vulkan compatible GLSL version 4.60. They are adapted from the magic number based bit interleaving code presented in [8]. These functions require the GLSL explicit arithmetic types extension to enable use of 64 bit integer types. The extension is enabled as show below:

```
#extension GL_EXT_shader_explicit_arithmetic_types_int64 : require
```

A.1 64 Bit 3D Morton Encode

```
uint64_t morton_magic_encode64(in uvec3 coord){
    const uint64_t magic[5] = {
        0xFFFF00000000FFFFUL,
        0x00FF0000FF0000FFUL,
        0xF00F00F00F00F00FUL,
        0x30C30C30C30C30C3UL,
        0x9249249249249249UL
    };
    const uint shift[5] = {32, 16, 8, 4, 2};

    u64vec3 result = u64vec3(coord);

    result = (result | (result << shift[0])) & magic[0];
    result = (result | (result << shift[1])) & magic[1];
    result = (result | (result << shift[2])) & magic[2];
    result = (result | (result << shift[3])) & magic[3];
    result = (result | (result << shift[4])) & magic[4];

    return(result.x | (result.y << 1) | (result.z << 2));
}
```

A.2 64 Bit 3D Morton Decode

```
uvec3 morton_magic_decode64(in uint64_t code){

    // See table at bottom of function
    const uint64_t magic[5] = {
        0x8208208208208208UL,
        0x00c00c00c00c00c0UL,
        0xf00000f00000f000UL,
        0x00000000ff000000UL,
        0xffff000000000000UL
    };
    const uint shift[5] = {2, 4, 8, 16, 32};

    u64vec3 result = u64vec3(
        (code >> 0) & 0x9249249249249249UL,
        (code >> 1) & 0x9249249249249249UL,
        (code >> 2) & 0x9249249249249249UL
    );

    u64vec3 masked;

    masked = result & magic[0];
    result = (masked >> shift[0]) | (masked^result);
    masked = result & magic[1];
    result = (masked >> shift[1]) | (masked^result);
    masked = result & magic[2];
    result = (masked >> shift[2]) | (masked^result);
    masked = result & magic[3];
    result = (masked >> shift[3]) | (masked^result);
    masked = result & magic[4];
    result = (masked >> shift[4]) | (masked^result);

    return(uvec3(result));
}
```

Appendix B

REQUIRED FEATURES AND EXTENSIONS FOR VULKAN AND GLSL

The following is a list of the optional features and extensions to both the Vulkan® API and GLSL language which our implementation requires to function. Support for these features and extensions may be dependent on both hardware and Vulkan implementation. Some Vulkan API features are available as extensions in earlier versions of the API.

Vulkan:

API version	Feature / Extension Name
1.2	`.bufferDeviceAddress`
1.2	`.descriptorBindingStorageBufferUpdateAfterBind`
1.0	`.shaderInt64`
1.0	`.sparseBinding`
1.0	`.sparseResidencyBuffer`

All code was written with GLSL 4.60, but may be compatible with earlier versions.

OpenGL Shading Language:

`GL_EXT_shader_explicit_arithmetic_types`	`GL_KHR_shader_subgroup_basic`
`GL_EXT_shader_atomic_int64`	`GL_KHR_shader_subgroup_vote`
`GL_EXT_buffer_reference`	`GL_KHR_shader_subgroup_arithmetic`
`GL_EXT_buffer_reference2`	

Although the framework itself does not require this feature, `VK_EXT_shader_atomic_float` is required by our PIC demonstration pipeline. It is assumed that many PIC simulation implementations would require some form of concurrent float arithmetic in its operations.

Atomic floating point operations are a very recent addition to the Vulkan ecosystem, and are currently supported only on a small percentage of devices. However, the extension is written to be vendor agnostic and atomic float arithmetic is supported through other compute only API on most vendor hardware. Thus it seems reasonable to assume that coverage of this extensions will eventually become similar to the other features already required by our implementation.